Chapter 1

System software

System software

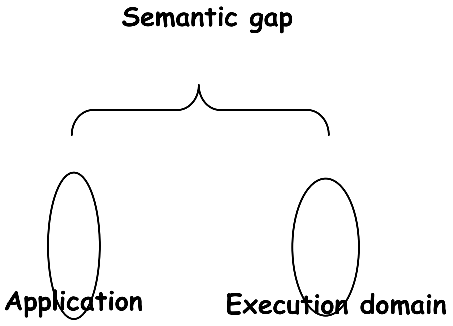
Need for Language Processing

Language processing activities arise due to the difference between the manner in which a software designer describes the ideas concerning the behavior of the software and the manner in which the ideas are implemented in a computer system.

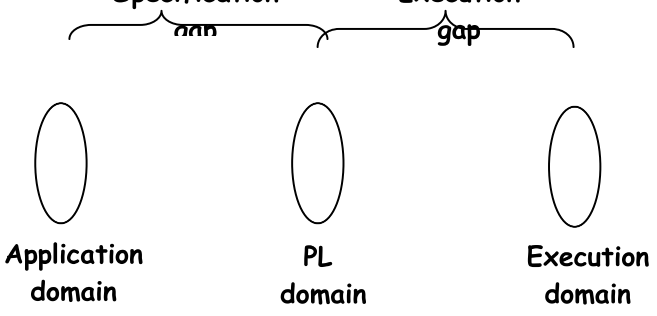
The designer expresses the ideas in terms related to the application domain of the software.

To implement these ideas, their description need to be interpreted in terms related to the execution domain of the computer system.

The term semantics is used to represent the rules or meaning of the domain and the term semantic gap to represent the differences between the semantics of the two domains. Figure below depicts the semantic gap between the application and execution domains.



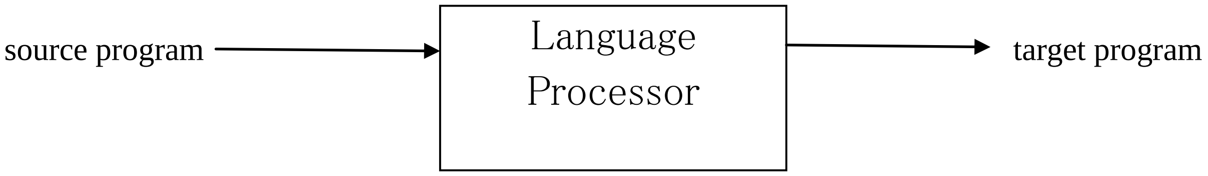
Consequences of semantic gap were large development time and efforts and poor quality of software. The semantic gap was made manageable with the introduction of programming language(PL) domain as shown in the figure below.



The gap between the application domain and PL domain is the specification gap. The specification gap is bridged by the software development team.
The gap between the PL domain and execution domain is the execution gap. The execution gap is bridged by the designer of the programming language processors.

Language Processor

Definition: A language processor is software, which bridges a specification or execution gap.
A spectrum of the language processors is defined to meet practical requirements.



The above figure shows a language processor that converts a given source language like C++ program to a required target program like machine language program.
A language processors like assemblers or compilers that bridge the execution gap are called Language Translators.

Types of Programmers

Application Programmers are the one that develop programs for solving a particular problem like accounting, word processing etc.
System Programmers are the one that develop programs that help application programmers in developing and executing their programs without concerning about the internal details of the system like compiler, macro processor, assembler, loader, linker, interpreter, editor and also operating system.

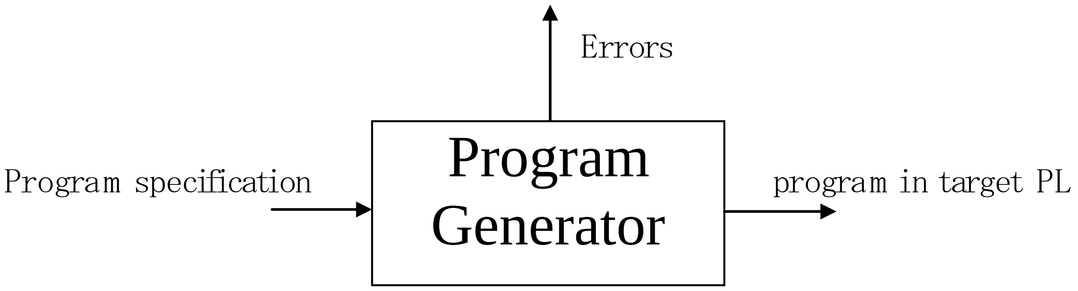
Language Processing Activities

There are two language processing activities

1. Program generation activity that bridges the specification gap.
2. Program execution activity that bridges the execution gap.

Program generation activity

Program generation activity deals with conversion of a given program specification into a program in the required language. This can be done with the help of program generators.
The program generator is a system software which accepts the specification of a program to be generated and generates a program in the target language.



Program execution activity

Program generation activity deals with conversion of a program in a PL into a executable program in the computer system. There are two models for this activity

1. Program translation.
2. Program interpretation.

Program Translation

Program translation model bridges the execution gap by translating a program written in a programming language called the source program into an equivalent program in the machine or the assembly language of the computer system called the target program.

Program Interpretation

Program interpretation model reads the source program and stores it in the memory. During interpretation it takes a source statement, determines its meaning and performs actions which implement it which would include the computational and input-output actions.

Fundamentals of Language Processing

Language Processing = Analysis of Source Program + Synthesis of Target Program.

A specification of the source language forms the basis of source program analysis. The specification consists of three components:

1. Lexical rules which govern the formation of valid lexical units in the source language.

2. Syntax rules that govern the formation of valid statements in the source language.
3. Semantic rules which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of source statement consists lexical, syntax and semantic analysis.

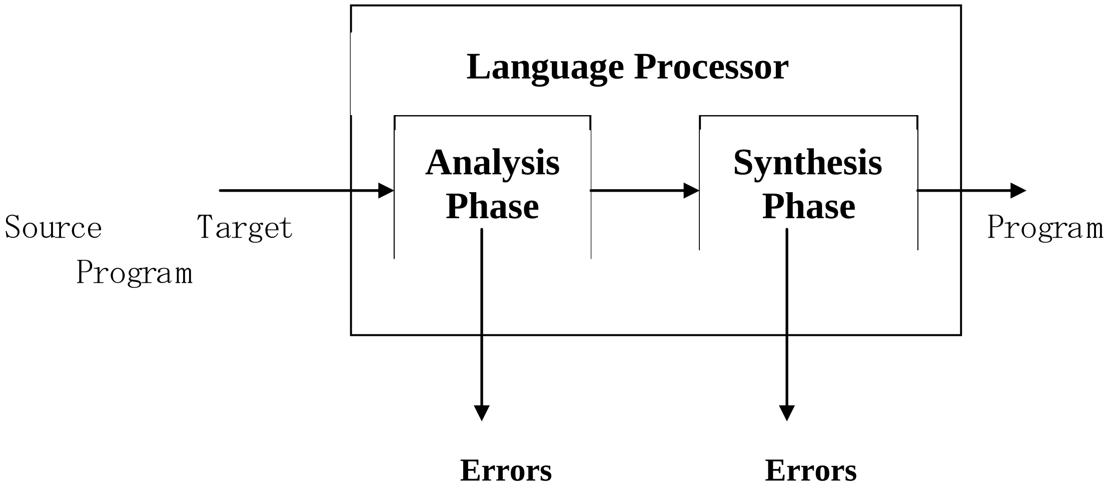
The synthesis phase is concerned with construction of target language statement(s) which have the same meaning as a source statement.

Typically, this consist of two main activities:

- Creation of data structures in the target program.
- Generation of target code.

Phases of a Language Processor

From the previous discussion it is clear that language processor consists of two distinct phases - the analysis phase and synthesis phase.



Types of Languages

High-level language

HLL allows a programmer to express algorithm in a more natural notation that avoids many of the details of how a specific computer functions. It also makes programming task simpler. But, we need a program to translate high-level language into a language that machine can understand. That means it needs a compiler. A compiler is a program that accepts a program written in a high-level language and produces an object program.

High level language is closer to the programmer than the hardware. Hence,

programmer can do programming without keeping any specific computer hardware in mind. In this sense, it is hardware independent. It is at the highest level of the program language hierarchy. These are machine independent and close to programmer and hence called high level languages. Program length is smaller than the corresponding low-level language program.
Example: PASCAL, C, C++, JAVA, FORTRAN, COBOL etc.

Assembly language

Assembly language is intermediate between machine code and high level. These are machine dependent. The assembly language programmer knows that computer has central processing unit (CPU) which has ALU, CU & few CPU registers.
It consists of series of instructions using mnemonic op-codes, addresses are symbolic, not absolute. It requires an assembler to translate a source program into object code. There is one to one correspondence between simple assembly language statements and machine language instructions. The following are the situations in which programmers would like to use assembly language

1. When interacting directly with the hardware.
2. When using processor specific instructions not available in HLL.
3. When no HLL exists.
4. When writing computer viruses, boot loaders and device drivers.

The program length is manageable but it is difficult to understand as compared to high level language program.
Example: Programming in 8085, 8086/8088, 8051 assembly language.

Machine language

It is a low-level machine dependent language and is at the lowest level of the program language hierarchy. The instructions in machine language are in a predefined instruction format specific to the machine. These instructions are a sequence of binary op-codes and data in machine specific format and hence are directly executed by the CPU.
The programming in machine level is very difficult and complex to understand and implement. Machine code can be regarded as primitive or cumbersome programming language. Hence the program length is also usually very long.
This language does not need any translator, it can be stored as a file on the disk. At the time of execution, machine language program has to be simply brought into memory from disk and then loaded at appropriate locations. This is done by loader.
Example
Assembly statement L 2,904 (0,1) in IBM 360 assembly language will have an equivalent machine language statement like 58201388H.

Types of System Software

System programs are the set of programs that are required for the effective execution of the application programs. Different types of system programs are as follows

Compiler

Compiler is a program that takes as input a program written in its source language which is a high level language (like C, C++) and produces an equivalent program written in its target language which is object code or machine code.

Interpreter

Interpreter is like a compiler with the difference that it executes the source program immediately rather than generating object code that is executed after translation is complete.

Assembler

An assembler is a language translator for the assembly language of a particular computer.

Linker

Both compilers and assemblers rely on linker which is a program that collects code separately compiled or assembled into a file that is directly executable. It also connects an object program to the code for standard library functions and to resources supplied by the OS of the computer such as memory allocators and I/O devices.

Loader

Compiler, assembler or linker will produce code that is not yet completely fixed to the memory address or ready to execute but whose principal memory references are all made relative to an undetermined starting location that can be anywhere in the memory. Such code is said to be relocatable and the loader will resolve all relocatable addresses relative to a given base or starting addresses.

Preprocessor

A Preprocesor is a separate program that is called by the compiler before actual translation begins. Such processor ignores and may delete comments, include other files and perform macro substitution.

Editor

Compilers usually accept source programs written using any editor that will produce a standard file such as ASCII file. Compilers normally are bundled together with editors and other programs into an interactive development environment (IDE).

Debugger

Debugger is a program that can be used to determine execution errors in a compiled program. It is often also packed with a compiler in an IDE. Running a program with a debugger differs from straight execution, in that the debugger keep track of most or all source code information such as line numbers, names of variables and procedures. It can also halt execution at prespecified locations called breakpoints as well as provide information on what functions have been called and what the current values of the variables are.

Operating System

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The operating system correspondingly includes programs to manage these resources such as traffic controller, a scheduler, memory management module, I/O programs and a file system.

Device driver

All the device dependent codes goes in the device drivers. Each device driver handles one device type or at-most one class of closely related devices. The device driver usually writes specific bit patterns to special locations in the I/O controllers memory to tell the controller on which device location to act and what actions to take.

Distinguish between assembler, compiler and interpreter

	Assembler	Compiler	Interpreter
(1)	It converts the assembly language program into its machine language equivalent.	It converts the source program written in HLL into its equivalent target code (machine language or assembly language).	It converts line by line of the source program written in HLL into its equivalent machine code in and executes it if the program is error free.
(2)	Speed of execution is	Speed of execution is	The speed of execution

	very fast.	fast.	is slow.
(3)	The translation is done of the entire program.	The translation is done of the entire program.	The translation program is done line by line.
(4)	It saves a lot of time as the program is to be assembled only once and can be executed repeatedly.	It saves lot of time as the program is compiled only once and can be executed repeatedly.	It takes a lot of time as for every run of the program it has to be translated and then executed.
(5)	It requires least memory space.	It requires a large memory space.	It requires less memory space.
(6)	It creates an object file.	It creates an object file.	It does not create an object file.
(7)	Software cost is very less.	Software cost is high.	Software cost is less.
(8)	Examples: Microsoft Assembler (MASM). Turbo Assembler (TASM).	Examples: MS-DOS C Compiler. TURBO C Compiler.	Examples: Basic Interpreter. SNOBOL

University questions

1. Explain the fundamentals of language processing.

CMPN May 04 (10M),May 05 (10M), May 07 (10 M)

2. Differentiate assembler, compiler and interpreter. IT May 05 (4M), Dec 05 (4M), May 07(5M)
3. Explain the various language processing activities to execute the program. CMPN Dec 2005 (8M)
4. What is system programming? List some system programs with their functions. CMPN May 10 (5 M)
5. What is system programming? Explain the various language processing activities to execute the program? CMPN Dec 05 (10 M)

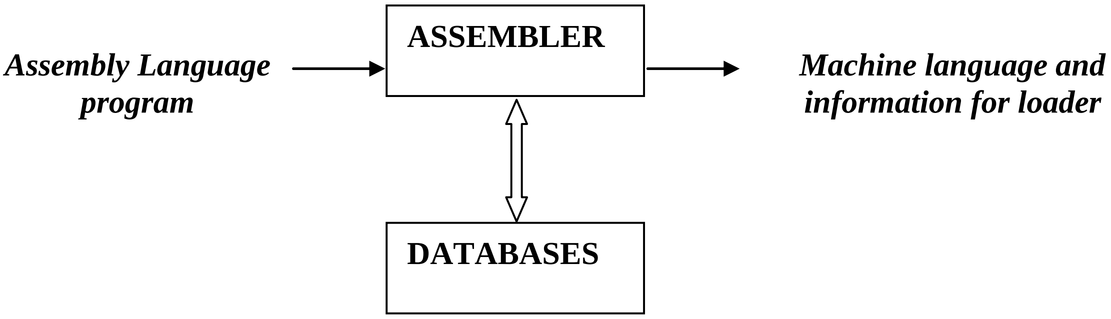
Chapter 2

Assemblers

Assemblers

Assembler

*Definition:*An assembler is a program that accepts as input an assembly language program and produces its machine language equivalent along with information for the loader. Figure below shows the function of an assembler.


Assembly Language program
ASSEMBLER
Machine language and information for loader
DATABASES

Assembly Language

Assembly language is a language in which each program statement corresponds to a single machine instruction that the processor can execute. This language is closer to the user as compared to the machine level language. It allows use of English like words to convey instructions. Hence it is easier for the user to program in this language as compared to machine language. It is particular for a certain processor and hence is machine dependent.

Every assembly language statement has the following format

[label] <opcode>   <operand>

Label if present associates a symbolic name with the memory word generated for the statement.

The opcode specifies the operation to be performed.

The operand specifies the operands(registers/memory locations/ ports) in which the operation needs to be performed.

Basic features of an assembly language program

1. **Mnemonic Operation Codes**: Instead of using binary op-codes, mnemonic’s are used for e.g. ADD, MOVE This mnemonic code provides a better understanding of the program and hence helps in developing or debugging a program.
2. **Symbolic Operand Specifications**: Operands can be specified in the form of symbolic reference rather than their absolute address or value.
3. **Declaring Data/Storage Areas**: Data constants can be declared in decimal notations or Hex notations. Even storage areas can be defined of a particular size. These data values & storage areas can be referenced by their symbolic name. e.g. DATA1 DC (05H), G DS 200

Types of statements in assembly language program

1. **Imperative statements:** These are **executable statements**, which are understood by the machine. An imperative statement indicates actions to be performed during execution of the assembled program. Each imperative statement exist, in the final m/c language program & is translated by the assembler in the format required by the machine.
   Example: MOV AX,BX

2. **Declarative statements:** These are **non executable statements.** They declare constants or storage areas in a program.
   Example: NUM DW 1020H

3. **Assembler directives**
   Statements of this kind neither represents m/c instruction to be included in the object program nor indicate the allocation of storage for constants or program variables. Instead these statements **direct the assembler** to take certain actions during the process of assembling the program.
   Example: DATA SEGMENT

Forward Reference Problem

Whenever a symbol is used before it is defined then while processing the operand field, the Assembler will look for the address of the symbol in its symbol table to replace it. However, this symbol is not to be found there.
The rules of writing an assembly language program only states that every symbol in the operand field has to be defined somewhere, in the same assembly program. Hence defining the symbol later is valid. The problem occurs because the definition of symbol occurs in the program after it is referenced. We call such a reference as "forward reference".

Every assembler needs to handle forward reference problem. There are two possible techniques to solve it

1. To scan the entire program two times. First scan (called first pass) will define all symbols and create symbol-table, second scan (called second pass) will complete the instruction having reference of symbol. This technique of designing is called **Two Pass Assembler design.**
2. To scan the entire program only once. If any instruction is found having reference to a symbol which is not yet defined then such statement is stored temporarily in a table of incomplete instructions (TII). These statements are completed when the symbol that they referred gets defined. This technique of designing is called **Single Pass Assembler design.**

Comparison of Two pass Assemblers and Single pass Assemblers

1. Two pass Assemblers are slower than Single pass Assemblers since they scan the source file two times.
2. Two pass Assemblers require less memory space than Single pass Assemblers since the memory allocated to first pass can be reallocated for the second pass.
3. Design of Two pass Assemblers is simpler than design of Single pass Assemblers due to modularity that is dividing a complex task into sub-tasks.

Design of Two Pass Assembler for IBM 360/370

Machine structure of IBM 360 and 370

1. Memory

The basic unit of memory in IBM 360 is a byte(8 bits) of information. The size of the address bus is 24 hence total space of IBM 360/370 memory is up-to 2^{24} bytes (16 MB). In each memory reference the unit of information transfer depends on the declaration/reference.

Unit of memory	Bytes	Length in bits
Byte	1	8
Halfword	2	16
Fullword	4	32
Double word	8	64

2. Registers

The IBM 360/370 has following registers

- 16 general purpose registers - 32 bit each
- 4 floating point registers - 64 bits each
- Program status word (PSW) - 64 bits

(PSW contains value of LC, protection information and interrupt status)

3. Instructions

The IBM 360/370 has arithmetic, logical, control/ transfer and special interrupt instructions. Register operands refer to the data stored in one of the 16 general registers which are addressed by 4-bit field in the instruction. Memory operands are referred by base displacement (optional index) addressing mode. The instructions are in one of the 5 instruction formats RR, ,RX, RS, SI, SS.

RR: Register-Register operand format.

RX: Register-Memory operand format.

RS: 2 Register-Storage operand format.

SI: Immediate-Storage operand format.

SS: 2 Storage operands format.

The formats are as below

RR format

op-code	R1	R2
8	4	4

RX format

op-code	R1	X2	B2	D2

8 4 4 4 12

RS format

op-code	R1	R3	B2	D2
8	4	4	4	12

SI format

op-code	I 2	B1	D1
8	8	4	12

SS format

op-code	L	B1	D1	B2	D2
8	8	4	12	4	12

4. PSEUDO Op-code

A pseudo op-code is an assembly language instruction that specifies an operation of the assembler.

List of Pseudo op-codes used in assembly language program (ALP)of IBM 360/370 are

(1)USING (2) DROP (3) DC (4) DS (5) EQU (6) START (7) END (8) LTORG

(1) USING: Using pseudo opcode indicates to the assembler which general register to use as a "BASE reg" and what its contents will be.
Example: USING *, 15

(2) DROP:Drop pseudo opcode indicates that a reg is not available as base reg.
Example: DROP 15

(3) START: Start pseudo opcode tells the assembler where the beginning of the program is and allows the user to give a name to program.
Example: JOHN START 1000

(4) END: End pseudo opcode tells the assembler, that the end of the program has reached (last card of the program)

(5) DC: DC pseudo opcode declares memory space for a symbol specified in the label field and converts the constant specified in the operand field. This constant is made available in the memory location at run time.
Example: FOUR DC F' 4'

(6) DS: DS pseudo opcode declares memory space like DC but there is no constant available in it at run time. This space contains garbage value initially when the program begins the execution.
Example: TEMP DS 100F

(7) EQU: EQU pseudo opcode equates a symbolic name to a value. This symbol is replaced by the given value whenever it is referenced.
Example: TEN EQU 10

(8) LTORG: Ltorg pseudo opcode allocates space for literals.

Purpose and function of two passes

Recollect that when a symbol is referred before its definition its called forward reference. This forward is solved by two pass assemblers by going through the program twice. The functions of both the passes are as follows:

PASS 1: Purpose :- Define symbols and literals.

To achieve this the first pass performs the following steps
Step 1: Keep track of location counter.
Step 2: Determine the length of machine instruction.
Step 3: Remember values of symbols until pass-2.
Step 4: Process some psuedo opcodes like EQU, DC, DS.
Step 5: Remember literals.

PASS 2: Purpose :- Generate the object program

To achieve this the second pass performs the following steps
Step 1: Look up the value of symbols.

Step 2: Generate instructions.
Step 3: Generate data.
Step 4: Process some psuedo opcodes like USING, DROP.

Relative symbols and absolute symbols:

If the value of the symbol changes with the change in the memory location of the instruction then the symbol is said to be relative.

If the value of the symbol remains constant irrespective of the change in the memory location of the instruction then the symbol is said to be absolute.

If we move the code from one memory area to another then the operands whose address changes are said to have relative address and whose address does not change are said to have absolute address.

Manual Assembly done using concept of a two pass assembler

(Q) Using any assembly language, write a sample assembly program, with respect to that program, describe how a 2 pass assembler will translate that. (18 M)

Example program in IBM 360/370 ALP

				FIRST PASS		*SECOND PASS*	
Stmt. #	Source Program			Rel. addr In HEX	Mnemonic inst.	Rel. Addr In HEX	Mnemonic inst.
1	JOHN	START 0		0		0	
2		USING	*, 15	0		0	
3		LR	1, FIVE	0	LR 1, __	0	LR 1, 5
4		A	1, FOUR	2	A 1, __	2	A 1, 10 (0, 15)
5		A	1, = F'2'	6	A 1, __	6	A 1, 18 (0, 15)
6		ST	1, TEMP	0A	ST 1, __	0A	ST 1, 14(0, 15)
7	FOUR	DC	F'4'	0E 10	4	0E 10	4
8	FIVE	EQU	5	14		14	
9	TEMP	DS	1 F	14	-	14	-
10		END		18	2	18	2
					1C		1C

Note: Length of instruction LR is 2 bytes (RR format)
Length of instruction A and ST is 4 bytes (RX format)

Databases needed for two pass assembler of IBM 360/370

1. Machine Operation table(MOT)

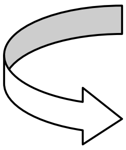
Mnemonic Opcode (4 Bytes)(Character)	Binary Opcode (1 Byte) (HEX)	Instruction Length (2 bits) (Binary)	Instruction Format (3 bits) (Binary)	Not used (3 bits)
"BALR"	5A	10	001	
–	–	–	–	–

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 - S1
100 - SS

2.Pseudo Opcode Table (POT)

Pseudo opcode (5 bytes) (Character)	Address of routine to process pseudo-up (3 bytes = 24 bit address) (HEX)
"DROP"	PTR DROP  Label of routine, actually address is present

3.Symbol table (ST)

Symbol (8 Bytes) (Character)	Value (4 bytes) (HEX)	Length (1 byte) (HEX)	Relocation (1 Byte) (Character)
"JOHNbbbb"	0	1	"R"

“FOURbbbb”	10	4	″R″
″FIVEbbbb″	5	1	″A″
“TEMPbbbb″	14	4	″R″

4.Literal table (LT)

Literal (8 bytes) (Character)	Value (4 Bytes) (HEX)	Length (1 Byte) (HEX)	Relocation (1 Byte) (Character)
“F’ 2’ bbbb”	18	4	″R″

5.Base table (BT)

	Availability Indicator (1 Byte) (character)	Designated relative address Contents of base reg. (3 bytes = 24 bit address) (HEX)
1	″N″	-
2	″N″	-
|		
|		
|		
|		
15	″Y″	000000

Databases required for PASS1

1. Input source program
2. Location Counter (LC) to keep track of each instructions location and to assign addresses fore each symbol define.
3. Machine Operation Table (MOT) indicates the symbolic mnemonic for each instruction and its length.

4. Pseudo Operation Table (POT) indicates for each pseudo opcode the action to be taken in pass 1.
5. Symbol Table (ST), to store each label and its corresponding value.
6. Literal Table (LT), to store each literal and its corresponding assigned location.
7. Copy of input to be used later by pass 2.

Databases required for Pass2

1. Copy of source program input from pass 1.
2. Location Counter (LC) for same purpose as pass 1.
3. Machine Operation Table (MOT) indicates for each instruction symbolic mnemonic, length, binary machine opcode and format.
4. Pseudo Operation Table (POT) indicates for each pseudo opcode the action to be taken in pass 2.
5. Symbol Table (ST) is prepared by pass1 containing all the labels and their corresponding values.
6. Literal Table (LT) is prepared by pass1 containing all literals and their assigned locations.
7. Base Table (BT) indicates which registers are current base registers and what are their contents.
8. Work-space INST is used to hold each instruction as its various parts are being assembled together.
9. Work-space PRINT LINE is used to produce a printed listing.
10. Work-space PUNCH CARD is used to punch the assembled instructions in the format needed by the loader.
11. An output deck of assembled instructions in format needed by the loader.

PASS 1 Algorithm

The purpose of first pass is to assign a location to each instruction and data defining pseudo instruction and thus to define values for symbols appearing in label fields of the source program.

1. Initially the location counter is set to the first location in the program (usually 0).
2. Source statement is read.
3. Opcode field is examined to determine if it is pseudo opcode. If yes, then special processing must be done corresponding to the pseudo opcode i.e. continue from step (5) but if not a pseudo opcode then continue from step (4).
4. Search in machine opcode table to find a match for the source statement' s opcode field.
    a. The matched MOT entry specifies the length of instruction (2,4,6 byte), which is stored in temporary variable"L" .
    b. The operand field is scanned for the presence of literal if found it is entered in literal table for literal processing.
    c. The label field is examined for the presence of the symbol if present it is saved in symbol table with current LC value.
    d. Value of LC is incremented by L.
    e. A copy of source code is saved for pass 2.

5. Special Processing for Pseudo Opcodes in Pass 1:

    DS/DC:

Adjust the LC before defining the symbol.

The operand field must be examined to determine the number of bytes of storage required and store it in L. Store symbol with current LC value in symbol-table and then update LC by L.

EQU:

The operand field value must be evaluated and assigned to the symbol in label field. This symbol along with its value is stored in symbol table.

USING / DROP:

Pass 1 is only concerned with pseudo opcodes that defines symbols or affect LC. USING and DROP do neither and hence these entries are only saved for pass 2.

LTORG:

Storage locations are assigned to the literals and entered in the Literal table.

END:

Pass 1 is terminated before transferring control to pass 2 storage locations are assigned to literals in the Literal table and the copy for pass 2 is rewind and reset.

6. Repeat from step (2) for each instruction.

PASS - 2 Algorithm

PASS 2 must structure the generated code into the appropriate format for later processing by the loader

1. Location counter is initialized as in pass 1.
2. A statement is read from the source file created by pass 1
3. As in pass1 the opcode field is examined to determine if it is a pseudo opcode if yes then special processing for that pseudo opcode is done i.e. continue from step (5), if not then continue from the next step (4).
4. Search in MOT

   The matched MOT entry specifies length, binary opcode and the format type of the instruction.

   The operand field is scanned and processing is done according to the format.

   RR format:

   Each of the register specification field is evaluated and the two fields are inserted into the respective 4 bit fields in the second byte of the RR instruction format.

   RX format:

   The register and the index fields are evaluated and the two fields are inserted into their respective 4 bit fields. The effective address of the operand is calculated and appropriate base and displacement is determined such that displacement is less than 4096. The base and displacement are inserted in 3rd and 4th bytes of RX instruction format.

   Instruction is assembled in the necessary format for later processing by the loader.

   Source listing, its assigned storage location and its hexadecimal equivalent is then printed.

   Location counter is incremented by length L.
5. **Special Processing of Pseudo Opcodes in Pass 2:**

   DS/DC:

   Processed same as in Pass 1, however actual code must be generated for DC pseudo opcodes depending upon the data type specified.

   EQU/START:

   Because symbol definition was completed in Pass 1. These cards are only required as a part of printed listing.

USING:

The operand fields of these pseudo opcodes are evaluated then the corresponding base table entry is marked as available.

DROP:

The corresponding base table entry is marked as unavailable.

LTORG:

Code should be generated for the literals in the Literal table.

END:

It indicates the end of source program and terminates the assembly process. Before terminating, code must be generated for any literals remaining in the Literal table.

6. Continue the processing of the next card from step (2).

University questions

1. State the reasons for the assembler to be multipass program.
   CMPN May 10 (5M) Dec 06(4M), May 05 (4M), IT May 04 (4 M)
2. Explain forward reference problem in assembler.
   IT Dec 07 (5M) May 05 (4M)
3. What is the forward reference problem? How it is handled in a two-pass assembler? Explain with the help of database.

IT May 07(10M)

4. What is assembly language? What feature of assembly language required us to build a to pass assembler?
   CMPN Dec 05(10M)
5. Explain with a neat flow chart and database, the working of a 2 pass assembler.
   CMPN May 08 (10M), Dec 08 (10M)
   IT Dec 08 (10M) Dec 07(10M), Dec 04(12M)
6. Using any assembly language, write a sample assembly program, with respect to that program, describe how a 2 pass assembler will translate that.
   IT May 04(10 M)
7. Explain with the help of databases each of the passes of 2-pass assembler.
   CMPN Dec 04(10M), IT Dec 05(10M)
8. Give the analysis and design of two pass assembler with respect to flow-charts data structures and algorithms.
   CMPN May 06 (20M),Dec 05(20M)
9. With reference to assembler explain the following tables with suitable example.
   1. POT 2. MOT 3.ST 4.LT
   CMPN May 10(10M)

Design of Single pass assembler for 8086/8088

Overview of 8086/8088 machine

1. Memory

The basic unit of memory in IBM PC (8086/8088) is a byte(8 bits) of information. The size of the address bus is 20 hence total space of memory is up-to 2^{20} bytes (1 MB). In each memory reference the unit of information transfer depends on the declaration/reference.

Unit of memory   Bytes Length in bits

Byte (DB)	1	8
Word (DW)	2	16
Double word (DD)	4	32
Quad word (DQ)	8	64
Tera word (DT)	10	80

2. Registers

The IBM PC 8086/8088 has following registers

- 4 general purpose registers - 16 bit each (can be used as upper and lower half 8 bit registers)
- 4 special purpose registers - 16 bits each (SP, BP, SI and DI)
- 4 segment registers - 16 bit each (having upper 16 bit of base address of 4 64K segments)
- Instruction Pointer (IP)
- Program status word (PSW) - 16 bits

3. Instructions

The 8086/8088 microprocessor has arithmetic, logical, control/ transfer and special interrupt instructions. Register operands refer to the data stored in one of the 4 general registers/ 4 special purpose registers which are addressed by 3-bit field in the instruction. Even when referred as one of the 8 8-bit registers they are addressed by 3-bit field in the instruction. In order to differentiate between 8 and 16 bit access the instruction has a single bit W (W=0 for byte and W=1 for word). Memory operands are referred by segment base address from segment registers and displacement/offset (16-bit) that helps in easy relocation of the program. The memory displacement can be specified using one of the addressing modes(direct, register indirect, relative, base index, relative base index). In 8086/8088 many mnemonics don't have any fixed length or format. The format is determined using operand field like JMP has 3 formats (short/near/far).

4. PSEUDO Op-code used in 8086/8088 ALP

A pseudo op-code is the mnemonic not understood by the machine (in microprocessor terms they are called assembler directives).

List of Pseudo op-codes used in 8086/8088 ALP are

Declarative statement Pseudo opcodes

(1) DB (2) DW (3)DD (4)DQ (5) DT

They declare memory space for data. The unit of transfer is as listed above (refer memory of 8086/8088).

Assembler Directive statement Pseudo opcodes

(1)SEGMENT : It indicates start of the segment. Name of the segment is specified in the label field.

Example: DATA SEGMENT

(2)ENDS : It indicates the end of the segment.

Example: DATA ENDS

(3)ASSUME : Since all memory addressing is segment directed for every symbolic reference the address has to determine its offset from the start of the corresponding segment to facilitate this the programmer has to do two things. One to load a segment register with the segment base and second to let the assembler know what address a segment register contains.

The second task is performed using the assume directive. It tells the assembler that it can assume that the address of the indicated segment is present in the named segment register.

Example: ASSUME DS: DATA, CS: CODE

(4)ORG : This pseudo op is used for manipulating the value of LC.

Example: ORG 1000H

(5)EQU : It declares symbolic name reference for the operand.

Example: ABC EQU 25

(6)END : Ends directive indicates the end of all segments.

Design issues in single pass assembler for 8086/8088

1.Forward reference.

Recollect that when a forward reference occurs then that forward reference is entered in a table of incomplete instructions (TII) by single pass assembler. When the label referred is defined this entry in TII could be analyzed and appropriate action can be taken to complete it. In 8086/8088 design we call this TII as **Forward Reference Table (FRT).**

In 8086/8088 every symbolic reference to a memory operand has to be assembled as segment base and offset. The segment base will be known using the owner segment field in **symbol table (SYMTAB)** followed by the register named in the ASSUME statement stored in a **segment register table (SRTAB).** When assembling a reference the assembler must know which segment register is currently available for specifying the segment base from SRTAB. However, this strategy would not work while assembling a forward reference. When the symbol is forward referenced which registers need to be used to assemble these references are not known as the assembler must know which segment register was available at the time of reference from old SRTAB. So as the assembler processes the source program it builds a table of SRTAB called **stored segment register table (STSRT).** Hence FRT table will contain STSRT number to identify the SRTAB to be used for the reference.

2.Length of instructions may be difficult to determine.

In 8086/8088, if forward reference is done then the space to be allocated for the instruction may not be know example for a destination for branch instruction the length of instruction cannot be determined since it depends on location of the symbol.

e.g. JMP NEXT

Length of instruction depends on whether NEXT is 128 bytes away from the instruction or more.

Such problems are overcome by considering default values such as for the instruction above the assembler will assume ***16-bit logical address*** until the user doesn' t specify a short displacement such as **JMP SHORT NEXT**

JMP NEXT • indicates 16-bit logical address for each instruction ***(Default).***

JMP SHORT NEXT • indicates that symbol is less than 128 bytes away from the instruction where the symbol is referenced.

Databases needed

1. Mnemonic Opcode table (MOT):

The table is hash organized and contains opcodes for assembly statements.
Alignment/Format information contains format of assembling the instruction if it is specific else specifies that there are multiple formats in which case assembler makes a default selection by calling the routine. The routine ID contains identifier of the routine which processes that particular opcode.

Mnemonic Opcode (6 Bytes)	Binary Opcode (2 Bytes)	Alignment/ Format Info (1 Bytes)	Routine ID (4 Bytes)

2. Symbol table (SYMTAB):

It is hash organized and contains all relevant information about the symbols defined and used in the source program.

Type field for EQU symbol indicates whether a symbol is to be given a numeric value or textual value, whereas for NON-EQU symbol it indicates alignment information.

The segment name in which owner is defined is given by the owner segment field which contains the SYMTAB entry number of the segment name.

Length field for storage operands specifies total number of bytes allocated to the operand, whereas for non-storage operands it specifies length of the instruction defined by the symbol.

Size field for storage operands specifies number of units declared for the operand, whereas for non-storage operands it specifies value associated with the symbol.

(8) (2)	(1)					(2)	(2)	(2)	(2)	(2)	(2)	(2)
Symbol	EQU	Seg name	defined	Type	Offset in seg	Owner seg	length	size	Source state #	Ptr to first FRT entry	Ptr to first CRT entry	Ptr to last CRT entry
1.CODE		Y	Y									
2.DATA		Y	Y									
3.	Y/N	Y/N	Y/N									
4.NEXT	N	N	Y			SYMTAB Entry #1						

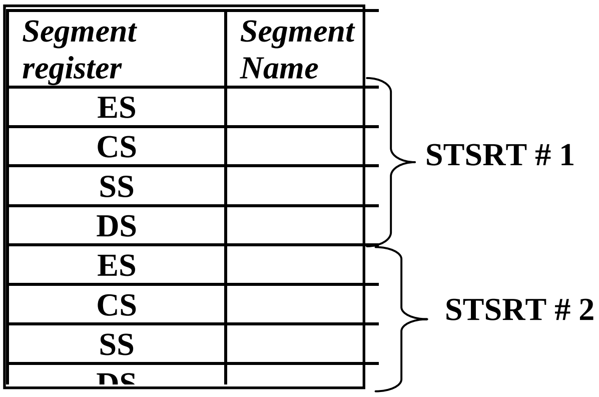
3. Segment register table (SRTAB)

It is created on every ASSUME statement.

Segment register	Segment Name
ES(00)	
CS(01)	
SS(10)	
DS(11)	

4. Stored segment register table (STSRT)

On every ASSUME statement the old SRTAB is sealed and a new SRTAB is created.



Segment register	*Segment Name*
ES	
CS	
SS	
DS	
ES	
CS	
SS	
DS	

5. Forward reference table (FRT)

This table contains information regarding forward reference in a linked list fashion. A field of SYMTAB entry points to the head of this list.

Entry# in STSRT - Stores current STSRT#

Instruction Address - Address of the statement in the source program which is to be completed.

Usage Code - This code is for the assembler to know how the incomplete statement is to be completed.

(2)	(1)	(2)	(1)	(2)
Ptr to next FRT entry	Entry in STSRT #	Instruction address	Usage code	Source statement #

6. Cross reference table (CRT)

At the end of assembly CRT is used for producing a cross reference directory. The directory lists all references to a symbol in the ascending order of statement numbers. Each SYMTAB entry points to the head and tail of a linked list. Each node of the list is an entry of CRT.

(2)	(2)
Pointer to next CRT entry	Source statement #

Flowchart of Single Pass Assembler for 8086/8088

SPARC (Scalable Processor architecture) assembler

SPARC assembler provides predefined section names.
For e.g.

. TEXT	executable inst.
. DATA	initialized read / write data
. RODATA	Read only data
. BSS	Uninitialized data areas.

- The programmer can switch between sections at any time in the source program by using assembler directives.
- Assembler maintains separate location counter for each named section. When it switches to a section it also switches to the counter associated with that section.
- By default, symbols used in a source program are assumed to be local to that program. However a section may freely refer to local symbols defined in another section of the same program.
- Symbols that are used in linking separately assembled programs may be declared as global or weak. A global symbol is either a symbol that is defined in the program and made accessible to others, or a symbol that is referenced in a program and defined externally.
- A weak symbol is similar to global symbol. The definition of a weak symbol may be overridden by a global symbol with the same name.
- Weak symbols may remain undefined when the program is linked, without causing an error.
- The object file written by the SPARC assembler contains translated versions of segments of a program and list of relocation & linking operations that need top be performed.
- Reference between different segments of the same program is resolved when the program is linked.
- The object program includes a symbol table that describes the symbols used during relocation and linking.

- SPARC assembly language has an unusual feature that is SPARC branch instructions are delayed branches.
- The instruction immediately following a branch instruction is actually executed before the branch is taken.

e. g. In the instruction sequence

```
CMP        % L0, 10
BLE        LOOP
ADD        % L2, % L3, % L4
```

ADD instruction is executed before the conditional branch BLE.
ADD instruction is said to be in the delay slot of the branch and it is executed whether or not the conditional branch is taken.
To simplify debugging, SPARC assembly language programmers often place NOP inst. in delay slots when program is written & code is later rearranged to move useful into the delay slot.

E. g.

```
LOOP: ADD % L2, %L3, % L4
      CMP % L0, 10
      BLE LOOP
      NOP
```

ADD instruction is logically first in the LOOP. It could be moved into the delay slot. This creates a problem; ADD should not be executed on the last execution of the loop.
The solution to this problem is that a conditional branch instruction is annulled; the instruction in its delay slot is executed if the branch is taken but not executed if the branch is not taken.

Eg.

```
LOOP:    CMP      % L0, 10
         BLE, A  LOOP
         ADD     % L2, % L3, % L4.   ← Delay slot
```

SPARC assembler provides warning messages to alert the programmer.

1. A label in the instruction in a delay slot usually indicates error.
2. A segment that ends with a branch instruction is indicated as a warning.

University questions

1. Explain with the help of flowchart and data structure of single pass Assembler.
   CMPN May04 (10M),May 05 (10M), May 06(10M), May 07 (10 M),Dec 07 (10M), IT May 05 (10 M)

2. Following Program is for 8086 processor. Give result for program after first pass and second pass of assembler with relative address of each instruction.

```
SAMPLE      START      4000
            USING      *, BX
            MOV        BL, NUM1
            ADD        BL, NUM2
            MOV        RESULT, BL
NUM1DC          10H
NUM2DC          20H
RESULT      DS         ?
            END
```

   IT May06 (10M)

3. Explain the SPARC Assembler.
   CMPN May04 (5M),Dec 04 (10M), May 06(6M), May 07(5 M), Dec 08 (5M), May 08 (10 M),IT Dec 07 (10M)

4. Explain the MASM Assembler.
   CMPN May 04(5M)

5. Explain the MASM and SPARC Assembler.
   CMPN May04 (10M),IT MAY 07(10M)

6. Discuss the merits and demerits of a one pass assembler over a multipass assembler. With examples, clearly state the important data bases required for a one pass assembler.

7. What is the forward reference problem? How is it solved in a one-pass assembler? Explain with the help of neat flowcharts the working of a one-pass assembler. Clearly indicate the organization of the databases in use.

8. What is the forward reference problem in a program ? Explain how it is solved in a one-pass assembler ? With the help of neat flowcharts explain the working of a one-pass assembler. Indicate and explain the Organization of various databases used during the assembly process.

Chapter 3

Macros & Macro-processors

Macros and Macro processors.

Need of Macros and Macro-processors

A programmer often finds some block of code in the program that he needs to type repeatedly. In this situation the programmer uses macros. Macros are single-line abbreviations for a group of instructions. The programmer types the block once and defines a single instruction to represent the block of code. For every occurrence of this one-line macro instruction in his program, the macro processor will substitute the entire block.

Comparison of Macros and Subroutines

1. A Macro call in the program is a single instruction that gets substituted by a group of instructions hence we say that macro-processor performs substitution as a process of expansion. A Subroutine call in the program is a single instruction that gets replaced by the address of the body of the subroutine after translation hence we say that subroutines are linked by linker.
2. Macro-processor performs code substitution before translation whereas linker performs Subroutine linking after translation. Hence the existence of macro is not known to the translator but the existence of subroutine call is defined by the translator as external references (to be linked by linker).
3. Since Macro substitutes the length of the final code increases whereas in Subroutine the block of code appears only once. Therefore a macro is preferred only if the block of code to be substituted is small, for large repeating block we must prefer a subroutine.
4. Macro-processor repeats same block of code for translator and hence the time of translation will increase if the block size is large. Linker links the subroutine by its address so the control of program execution must transfer at run-time. The overhead of control transfer is high if the block size is small.

Example Macro in Assembly Language

Consider the following program -

```
             .
             .
        A    1,    DATA
        A    2,    DATA
        A    3,    DATA
             .
             .
             .
        A    1,    DATA
        A    2,    DATA
        A    3,    DATA
             .
             .
             .
DATA  DC    F ‘5’
```

In the above program the sequence

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

is repeated multiple times.

A macro facility permits us to attach a name to this sequence and use the name in its place.

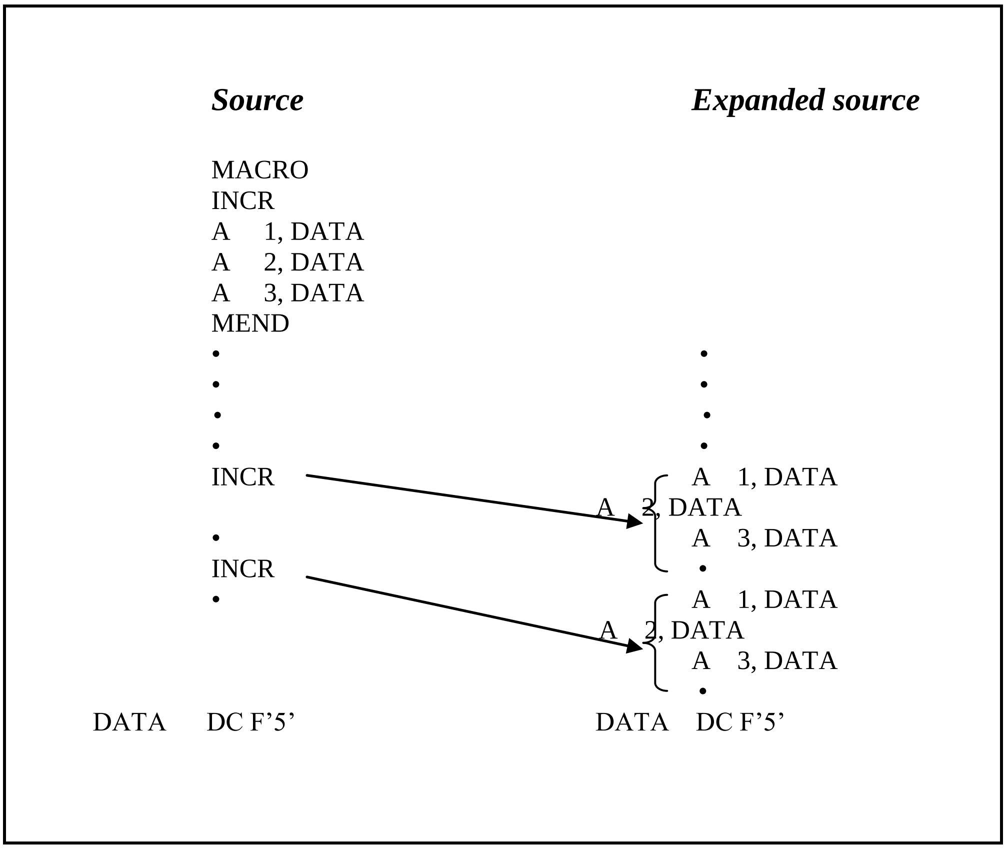
MACRO DEFINITION

```
MACRO                    Macro Header
[       ]                Macro name
_____
_____
            Sequence to be abbreviated
_____

MEND               End of definition
```

- MACRO and MEND are macro pseudo opcodes.
- The line after the pseudo opcode MACRO is a macro prototype.
- The occurrence of macro name in the source program as an operation mnemonic to be expanded is called a macro call.

- The process of replacement of macro call by the instructions is called expanding the macro
- Macro definition does not appear in the expanded source code.



```
            Source                          Expanded source

            MACRO
            INCR
            A    1, DATA
            A    2, DATA
            A    3, DATA
            MEND
            •                                •
            •                                •
            •                                •
            •                                •
            INCR                             A    1, DATA
                                         A   2, DATA
            •                                A    3, DATA
            INCR                             •
            •                                A    1, DATA
                                         A   2, DATA
                                             A    3, DATA
                                             •
DATA        DC F'5'                  DATA    DC F'5'
```

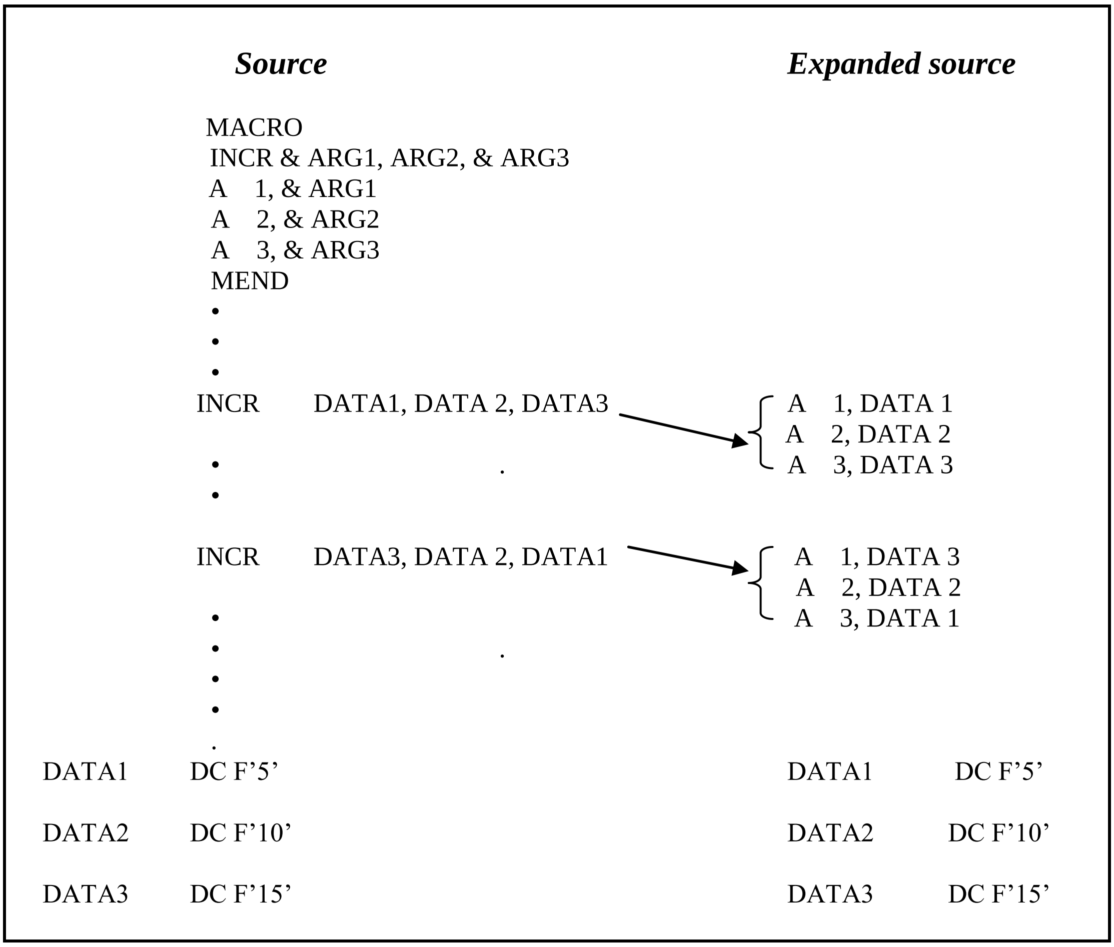
Features of Macro-processor / Macro Facility

A macro-processor can offer the following facility

1. Macro Instruction Arguments.
2. Conditional Macro.
3. Expansion-time loops in Macro.
4. Macro calls within Macro / Nested Macro calls.
5. Macro Definition within Macro.

1. MACRO INSTRUCTION ARGUMENTS

Providing arguments to macro call.



```
           Source                                   Expanded source

          MACRO
          INCR & ARG1, ARG2, & ARG3
          A   1, & ARG1
          A   2, & ARG2
          A   3, & ARG3
          MEND
          •
          •
          •
          INCR     DATA1, DATA 2, DATA3              A   1, DATA 1
                                                     A   2, DATA 2
          •                .                         A   3, DATA 3
          •

          INCR     DATA3, DATA 2, DATA1              A   1, DATA 3
                                                     A   2, DATA 2
          •                                          A   3, DATA 1
          •                .
          •
          •
          .
DATA1     DC F'5'                                    DATA1      DC F'5'

DATA2     DC F'10'                                   DATA2      DC F'10'

DATA3     DC F'15'                                   DATA3      DC F'15'
```

- Parameters like &ARG1, &ARG2 and &ARG3 are called macro instruction arguments or dummy arguments or formal parameters. Parameters specified in the Macro Call like DATA1 , DATA2 and DATA3 are called actual parameters.
- Macro language symbol is distinguished from assembly language symbol by the ampersand (&)
- It is possible to supply more than one argument in a macro call.
- There are two types of arguments:
  1. Positional arguments: Arguments are matched with dummy arguments according to the order in which they appear.
  2. Keyword arguments allow reference to dummy arguments by name and associate them with defaults. *(Refer Class Notes for Example)*

2. CONDITIONAL MACRO EXPANSION

A macro definition that is able to generate different instruction sequence when different relations hold between the actual parameters of a call is a conditional macro.

Labels starting with a period (.) such as .AGAIN are macro labels and do not appear in the output of the macro processor.

Pseudo opcodes to be supported for Conditional Macro

AIF: Conditional branch pseudo opcode

Syntax: *AIF (condition) macro label*

If the condition is true then control is transferred to the statement pointed by the macro label else continue with the next statement.

AGO: Unconditional branch pseudo opcode

Syntax: *AGO macro label*

When AGO is encountered control is transferred directly to the statement pointed by the macro label

ANOP: It is simply used to define a sequencing symbol(i.e. a macro label).

```
              Source                                        Expanded source
              MACRO
          VARY  & ARG1, & ARG 2, & ARG 3, & COUNT
& ARGO    A    1 ,  & ARG1
          AIF  (& COUNT EQ 1)  . FINI
          A    2,  & ARG2
          AIF ( & COUNT EQ 2)  . FINI
          A   3,  & ARG3
.FINI     MEND
              •
              •
              •
          VARY DATA1, DATA2, DATA3, 3                        A    1, DATA 1
              •                                              A    2, DATA 2
              •                                              A    3, DATA 3


          VARY  DATA3, DATA2, 2                              A    1, DATA 3
              •                                              A    2, DATA 2


          VARY DATA3, 1                                      A    1, DATA 3
```

3. EXPANSION TIME LOOPS

These are loops that are processed at the time of expansion and not at the time of execution. They do not exist in the expanded code.

Pseudo opcodes for expansion time loops in macro

LCL: It is used to declare a local expansion time variable and is always initialized to zero.

Example *LCL &A* declares & A is an expansion time variable local to the macro and is initialized to zero.

GBL: It is used to declare a global expansion time variable. It exists across all macro calls situated in a program.

Example *GBL &A*

SET: It sets the expansion time variable occurring in its label field to the value of the expression occurring in the operand field

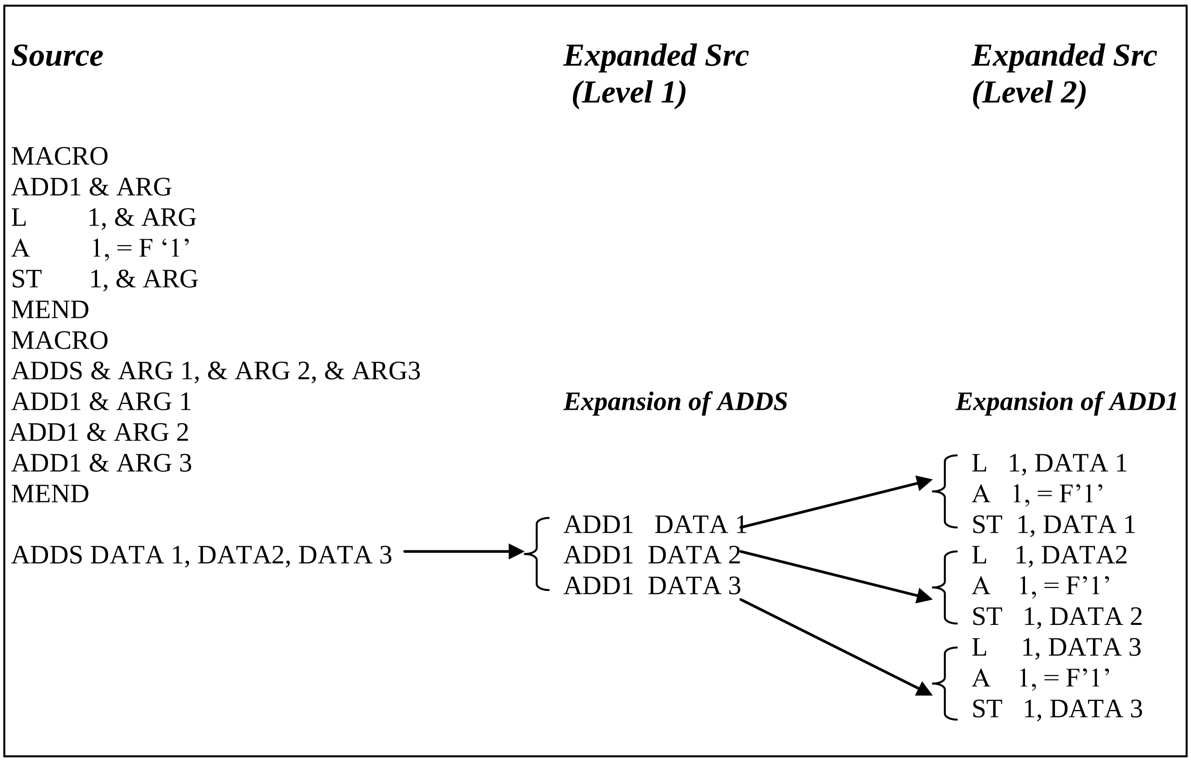
Example *&A SET &A + 1*

Increments the value of &A by 1

(For Example Refer Class Notes)

4. MACRO CALLS WITHIN MACROS / NESTED MACRO CALLS

No special Pseudo opcode for this feature.



Source

```
MACRO
ADD1 & ARG
L       1, & ARG
A       1, = F ‘1’
ST      1, & ARG
MEND
MACRO
ADDS & ARG 1, & ARG 2, & ARG3
ADD1 & ARG 1
ADD1 & ARG 2
ADD1 & ARG 3
MEND

ADDS DATA 1, DATA2, DATA 3
```

Expanded Src (Level 1)

Expansion of ADDS

```
ADD1   DATA 1
ADD1  DATA 2
ADD1  DATA 3
```

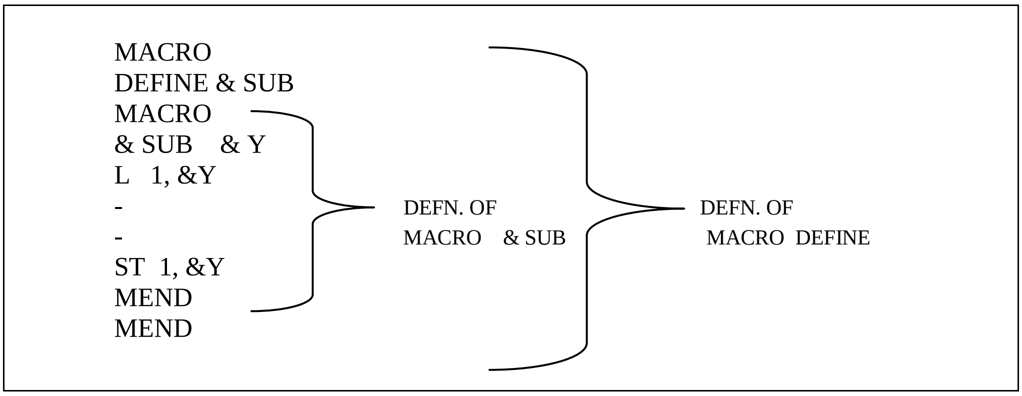
Expanded Src (Level 2)

Expansion of ADD1

```
L   1, DATA 1
A   1, = F’1’
ST  1, DATA 1
L    1, DATA2
A    1, = F’1’
ST   1, DATA 2
L     1, DATA 3
A    1, = F’1’
ST   1, DATA 3
```

5. MACRO INSTRUCTIONS DEFINING MACROS

A single macro instruction might be used to simplify the process of defining a group of similar macros.



```
MACRO
DEFINE & SUB
MACRO
& SUB   & Y
L  1, &Y
-
-
ST  1, &Y
MEND
MEND
```

This example defines a macro instruction DEFINE which when called with a subroutine name defines a macro with the same name as subroutine (&SUB)

The user might call the macro DEFINE with the statement

```
       DEFINE COS
```

defining a new macro named COS

Then the user might call the COS macro as follows :-

```
      COS ABC
```

Phases of Macro-processor

1. Macro Definition Phase: There are two functions to be achieved in this phase. They are
   - Recognize macro definition: Macro-processor must search for pseudo opcode MACRO to detect a macro definition.
   - Save the definitions: Once found it must define macro name in MNT(Macro Name Table) and model statements (till MEND) in MDT(Macro Definition Table) to be used for expansion.
2. Macro Expansion Phase: There are two functions to be achieved in this phase also. They are
   - Recognize calls: The Macro-processor must search the mnemonic field in MNT to detect macro calls
   - Expand calls and substitute arguments: On detecting macro call the macro-processor must substitute it from MDT.

Design of two pass Macro-processor

The two pass design will perform two scans over the source to perform the two phases and then generate the expanded source code.

Databases needed

1.Argument list array (ALA)

ALA is used during PASS 1 as well as PASS 2.

During PASS1: ALA is used to simplify later argument replacement during macro expansion. Dummy arguments in the macro definition are replaced by positional indicators/index markers when the definition is stored. ith dummy argument is represented as # i.

Example

```
MACRO
       INCR & ARG1, & ARG2, & ARG3
        A 1, & ARG 1
       A 2, & ARG 2
       A 3, & ARG 3
       MEND

      INCR DATA 1, DATA 2, DATA 3
```

Argument list array

Index	***Arguments*** ***(8 bytes per entry)***
0	***“bbbbbbbb”***
1	**“&ARG 1 *bbb*”**
2	**“& ARG2 *bbb*”**
3	**“&ARG 3 *bbb*”**

Due to ALA the MDT will appear as follows:

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

During PASS 2: Macro call arguments (actual parameters) are substituted for index markers stored in macro definition using ALA This list will be used while expanding a call which uses keyword parameters.

Argument list array

Index	***Arguments*** ***8 bytes per entry***
0	***“bbbbbbbb”***

1	**"DATA 1 *bbb*"**
2	**"DATA 2 *bbb*"**
3	**"DATA 3 *bbb*"**

2. Macro definition table (MDT)

Every line of each macro definitions, except the MACRO header is stored in MDT. MACRO header is useless during macro expansion so is not stored, MEND is used to indicate end of macro during expansion. The prototype is stored as it is to facilitate keyword argument replacement.

Macro definition table		
Index	Card 80 bytes per entry	
–		–
–		–
15		INCR & ARG 1, & ARG2, &ARG 3
16		A 1, #1
17		A 2, #2
18		A 3, #3
19		MEND
–		–

3. Macro name table (MNT)

Macro name is used to search for Macro calls and MDT Index refers to MDT entry from where definition begins for the specific Macro.

Macro name table		
	8 bytes	***4 bytes***
Index	***Name***	***MDT Index***
-	**-**	**-**
3	**"INCR *bbbb*"**	**15**
-	**-**	**-**

Databases required for Pass1

1. The input macro source deck
2. The output macro source deck copy for use by PASS 2
3. MDT to store body of macro definition.
4. MNT to store names of defined macros
5. MDTC (macro definition table counter) used to indicate next available entry in MDT
6. MNTC (macro name table counter) used to indicate next available entry in MNT.

7. The ALA used to substitute index markers for dummy arguments before storing a macro definition.

Databases required for Pass2

1. The copy of input macro source deck.
2. The output expanded source deck to be used as input to the assembler.
3. MDT, created by PASS 1
4. MNT, created by PASS 1
5. MDTP (Macro definition table pointer) used to indicate the next line of text to be used during macro expansion.
6. ALA is used to substitute macro call arguments for the index markers in the stored macro definition.

PASS 1 Algorithm (Macro Definition Phase)

(1) Read the statement from the card. If it is END pseudo opcode then go to PASS2 else if it is a MACRO pseudo-op continue with (i) to (v) else repeat step1.

i. Read the next statement.

ii. The first line of the definition is the macro name line. The name is entered into the MNT along with the pointer (MDT index) to the first location of the MDT entry from where the definition starts.

iii. The entire macro definition that follows is saved in the next available locations in the Macro Definition Table (MDT).

iv. Reading proceeds from the source card; as each successive line is read, dummy arguments in the macro definition are replaced with positional indicators and then stored in the MDT along with the MEND statement.

v. On encountering MEND, go to step 1

Note: *When the END pseudo-op is encountered, all of the macro definitions have been processed so control transfers to pass2.*

Flow-chart of Pass1

PASS 2 Algorithm (Macro Expansion Phase)

The algorithm for pass2 tests the operation mnemonic of each input line to see if it is a name in the MNT.

(1) Read the statement from the card. If END pseudo-op is encountered, the expanded source deck is transferred to the assembler for further processing else if it is a macro call (mnemonic found in MNT) go to step (i) to (iv) else repeat step1

i) The macro-processor sets a pointer, the MDTP (Macro Definition Table Pointer) to the corresponding macro definition stored in the MDT which is obtained from the MDT index field of the MNT entry.

ii) The macro processor then prepares the ALA to replace the index markers by the actual parameters.

iii) Reading proceeds from the MDT; as each successive line is read, the index markers from MDT are replaced by actual parameters in the macro definition.

iv) Reading of the MEND line in the MDT terminates expansion of the macro and go to step (1)

Flow-chart of PASS 2

Design of single pass Macro-processor

A single pass Macro-processor will scan the source only once. A forward reference on Macro call cant be handled as the size of macro is not known. So in single pass design all macro definitions must be in the start. Therefore the first statement of source program must be the MACRO header. After the end of first macro (that is the statement after MEND) is the macro header of second macro and so on. If after any MEND statement the next is not MACRO then it stops the Macro definition phase and starts Macro expansion phase.

Databases needed

1. Macro name table (MNT)

Macro name is used to search for Macro calls and MDTP refers to MDT entry from where definition begins for the specific Macro. No of #PP, #KP, #EV's specify number of Positional parameters, Keyword parameters and Expansion-time variables respectively. KPTP refers to KPT entry from where keywords are defined with their default values (if any).

Example:

```
 MACRO
 EVAL1  &X, &Y, &OP = ADD
  LOAD   &X
 &OP     &Y
 STORE   &X
 MEND
–
EVAL1  P,Q
–
EVAL1  R,S,OP=SUB
–
```

Macro Name	*Pointer To MDT (MDTP)*	*No. of Positional Parameters (#PP)*	*No. of Keyword Parameters Table (#KP)*	*Pointer to keyword Parameter Table (KPTP)*	*Number of expansion Time variables (#EV's)*
-	**-**	**-**	**-**	**-**	**-**
EVAL1	**26**	**2**	**1**	**6**	**0**
-	**-**	**-**	**-**	**-**	**-**

2. Macro definition table (MDT)

Like in MDT of two pass design the parameter are replaced by index numbers. *Note that separate index markers will be used for expansion-time variables so as to differentiate them from formal parameters (Positional and Keyword).*

Macro definition table (MDT)

```
           EVAL1  &X, &Y, &OP = ADD
           LOAD    #1
          #3        #2
          STORE   #1
          MEND
```

3. Keyword parameter table (KPT)

For the keyword parameters in the macro KPT is prepared which would be used to the store the default value which would be used if they are not passed when the macro is called.

	Keyword Parameter Table	
	Keyword	Default
#6	OP	ADD

4. Actual parameters list (APL)

For every macro call a new APL is created. It contains actual parameters corresponding to the formal parameters in the MDT. The list is created using the parameters passed in the macro call and using defaults (when actual parameters for keywords are not passed in the call)from KPT.

APL

#1	P
#2	Q
#3	ADD

5. Expansion time variable storage (EVS)

EVS is used to store the expansion time variables. When a macro call occurs to a macro having expansion-time variables, EVS is allocated for all local variables used in the macro. All the locations are initialized to 0 at the start of every expansion.

Expansion-time Variable Storage

(EVS)

#EV1 Value of expansion-time variable 1 #EV2 #EV3
Value of expansion-time variable 2
Value of expansion-time variable 3

6. Macro expansion counter (MEC)

Macro expansion counter points to the statement in MDT being processed at the time of macro expansion.

MEC

26

<u>Algorithm for Single Pass Macro Processor</u>

(1) Read the statement from the card. If it is a macro definition i.e. MACRO pseudo-op then continue macro definition else go to (2)

Macro definition:

i) Insert macro name, parameter information, pointer value of MDT from where the definition starts and number of expansion-time variables in Macro Name Table (MNT).

ii) Store the entire macro definition in the Macro Definition Table (MDT).

iii) On MEND, store in MDT and go to (1)

(2) Check whether it is a macro call i.e. mnemonic field matches with name in MNT. If found continue with Macro expansion else go to (3)

Macro expansion:

i. Obtain information from MNT regarding position of macro definition in MDT and copy it into MEC. Create APL from parameters specified in the macro call and defaults for keywords. Allocate space for Expansion-time Variable Storage based on number of expansion-time variables.

ii. For all statements from MDT pointed by MEC perform the following:

- The unconditional branch statement AGO will enforce changes in the normal sequential order.
- The conditional branch statement AIF will evaluate condition based on the actual parameters from APL and values from EVS known at expansion-time and will enforce changes in the normal sequential order only if the condition is satisfied.
- ANOP statement is a dummy statement and so MEC increments by 1 to go to next statement.
- SET statement modifies values of expansion-time variable specified in label field based on operand specification.
- MEND statement stops the macro expansion.
- If it is any other model statement then replace the index markers by actual parameters using APL or by values from EVS. Generate the statement in expanded source.

(3) Write the statement for expanded source and if END is encountered supply the expanded source for assembly processing else read the next statement from source and go to step (2) .

Flowchart (Without nested macro calls)

Flowchart (With nested macro calls)

Design of a macro assembler

- Use of a macro pre-processor followed by a conventional assembler is as expansive way of handling macros.
- The number of passes over the source program is large and many functions get duplicated.
- If macros are handled by a macro assembly which performs macro expansion and program assembly simultaneously, then many of the functions could be merged and numbers of passes will be reduced.

Pass structure of a macro assembler

PASS 1

* Macro definition processing : Construct MNT, MDT and KPT

* Collect declared symbols and related information

PASS II

* Macro expansion
* Location counter processing and completing SYMTAB information
* Processing of literals
* Intermediate code generation

PASS III

* Generation of target code

Pass II has many functions to perform. Hence its code size would be considerable. Since it includes both macro expansion and PASS I of conventional assembler, all the data structures need to exist during this pass. Storage requirements for Pass II would thus determine the requirements of the macro assembler as a whole.
Pass I and Pass II can be combined if all macros are defined at the start.

Macro Assembler flowchart:

(Refer class notes)

<u>University questions</u>

1. Define Macro. How it is different from subroutine?

IT Dec04 (05M)

2. What are the advantages of using macro in place of subroutine and when will a macro be used. Explain?

CMPN Dec04 (10M), May05 (10M), Dec06(10M)

3. State difference between macro and subroutine and explain when a macro will be used.

IT May07 (10M)

4. Explain when will a macro be used in a program. How is macro different from subroutine.

CMPN May10 (10M)

5. Write short note on ANSI C Macro language.

CMPN May07 (5M), Dec07 (5M)

6. Describe the features offered by Macro Facility. Give example. CMPN May08 (10M), IT Dec05 (10M).

7. Explain the recursive macro expansion and nested macro calls with the help of example.

CMPN May04 (10M) May05(10M)

8. Explain conditional macro expansion with suitable example.

CMPN May06 (06M)

9. State different features offered by macro. Describe the use of the following in a two pass macro-preprocessor

i) Macro Name Table
ii) Macro Definition Table
iii) Argument List Array
iv) Macro Expansion Counter

IT May04 (10M)

10. Explain macro and database for two pass macro.

IT May06 (10M), Dec07 (10M)

11. Explain two pass macro processor with neat flowcharts and databases.

CMPN Dec07 (10M), IT May05 (8M)

12. Explain one pass macro-processor with databases.

IT Dec04 (5M)

13. Explain the single pass macro processor with the help of flowchart.
CMPN Dec04 (10M)

14. Explain with the help of flowchart and databases working of one pass macro processor.

CMPN Dec08 (10M)

15. Explain the design of macro processor.

CMPN May04 (10M)

16. State the different features offered by a macro. Describe the use of the following

1. MNT 2. MDT 3. APL 4. MEC

IT May 04 (10M)

17. Write the detailed note on macro processor.

CMPN Dec05(10M)

18. Explain design of one pass macro processor to handle nested macro calls. What are the different databases needed? Explain.

CMPN May06 (10M), May07 (10M)

19. Explain single pass algorithm for macro definition within macro. CMPN Dec07 (10M), May08(10M)

20. Write short note on Macro Assembler.

CMPN May 07 (5M), Dec07 (5M), May08 (10M)

Chapter 4

Loaders and Linkers.

Loaders and Linkers

A loader is a program which accepts the object program decks, prepares these programs for execution by the computer and initiates execution.

Basic functions of a loader

1) Allocation: *Allocate space in memory for the program.*
a) Translator cannot allocate since modules may overlap or large wastage of memory may occur.
b) Programmer cannot allocate due to the same reasons.
Hence loader can obtain information of the length of the object program from the translator and thereby allocate memory areas to every subroutine without causing overlap or wastage of space.

2) Relocation: *To adjust all address dependent locations like address constants to correspond to the allocated space.*
Since allocation is performed by loader, information regarding starting address will not be available to the translator hence translator will be able to provide only a relative address i.e. offset of the symbol from the start of program.
The loader when decides the starting address, would need to change the relative address in the instruction to absolute address.
This process is termed as relocation.
The information about address sensitive instructions need to be given by the translator to the loader so that it can perform relocation of address sensitive instructions.

3) Linking: *Resolves symbolic reference between object decks/text.*

It is difficult for translators to link the subroutines since different programs are written in different languages and use different translators.

Hence the information that a particular subroutine is referenced by a program is passed by translator to the loader.

Considering the reference as an external symbolic reference, if all the translators use common schemes then the loader will be able to assign absolute address of the subroutine whenever it is referenced by another subroutine.

This process is termed as linking.

4) Loading: *Physically places the machine instructions and date into the memory.*

If loading is done by translator then

a) There will be wastage of memory as translator occupies memory.

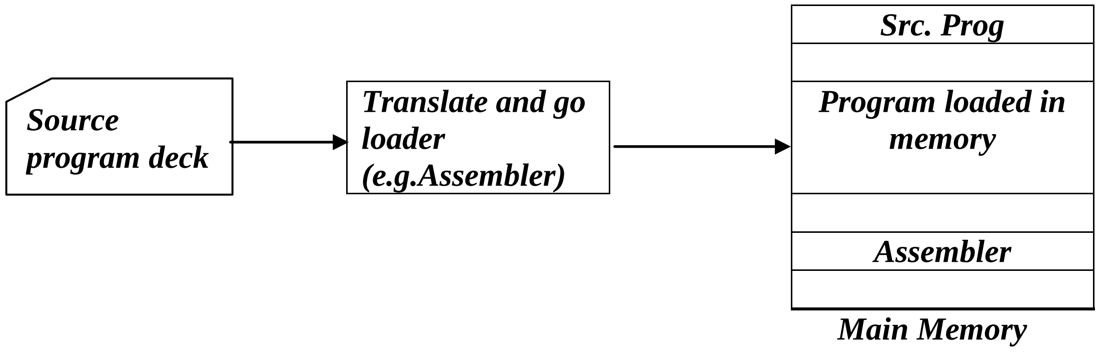
b) Multiple languages support will be difficult.

c) Program needs to be re-translated every time it is run.

Hence loading must be performed by a dedicated loader as in general loader scheme.

Loader schemes

1. Translate and go loader / Compile and go loader / Assemble and go loader



a. In this scheme there is no loader, the translator itself loads the assembled program into main memory.

b. The assembler runs in one part of the memory and places the assembled machine instructions and data as they are assembled directly into their assigned memory locations.

c. When assembly is completed, the assembler causes a transfer to the starting instruction of the program.

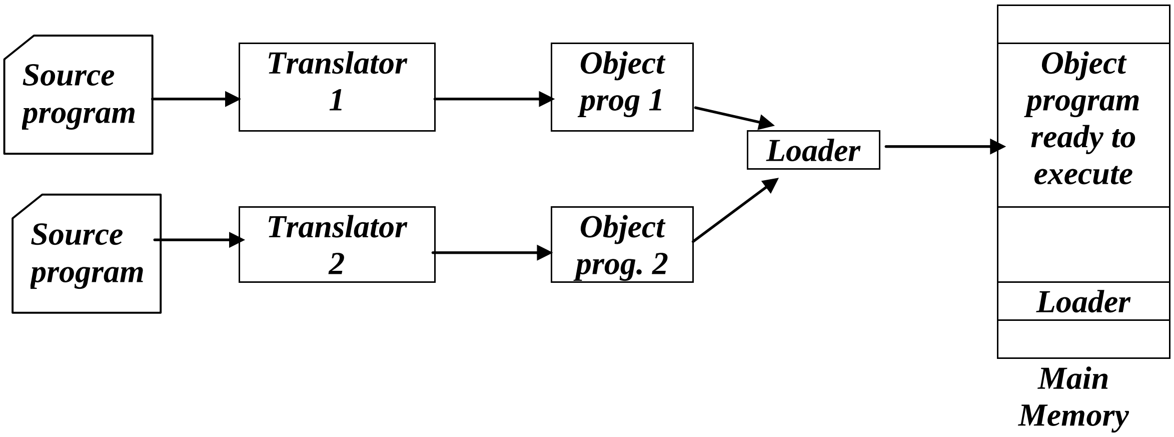
Disadvantages:

a. Portion of memory is wasted because core is occupied by the translator is unavailable to the object program.

b. It is necessary to re-translate the user' s program every time it is executed.

c. It is very difficult to have multiple language support especially when a single source program is in different languages (mixed language).

2.General loader scheme



All disadvantages of translate and go loader are overcome using this scheme since

a. Loader is smaller than translator hence memory is not wasted much.

b. Re-translation is not necessary every time the program is executed since object decks are used.

c. Even If all source program translators are incompatible, still they can use compatible conventions for object program deck format. Hence, it is possible to write subroutines in different languages since the object deck to be processed by the loader will be in the same machine language and with same format.

There are different types of loaders depending on the functions performed by the general loader.

Types of Loader

1. Absolute Loader

It is the simplest of all loaders as it only performs loading function.

a) Translator outputs machine instruction and data of source program which is punched on cards forming an object deck.

b) The loader simply accepts the machine language and places it into the core as prescribed by the assembler.

c) The loader then initiates execution by transferring control to the starting of the program.

Four Loader functions are accomplished as follows:

Allocation	-	by programmer
Linkage	-	by programmer
Relocation	-	by assembler
Loading	-	by loader

Advantages:

a. More core is available to the user since translator is not in memory.

b. Absolute loaders are simple to implement.

Disadvantages:

a. Programmer must specify the address in the core where the program is to be loaded and hence overlapping may occur or wastage of core may occur.

b. If there are multiple subordinates programmer must remember the address of each and use the absolute address explicitly in other subroutines to perform subroutine linkage.

Example

Assume that there are two functions MAIN and SQRT. The function MAIN calls the function SQRT. The programmer binds the two modules at address 100 and 400 respectively (assuming that the MAIN function will occupy 300 bytes space). Address of SQRT is specified in the assembly language program, hence all addresses absolute.

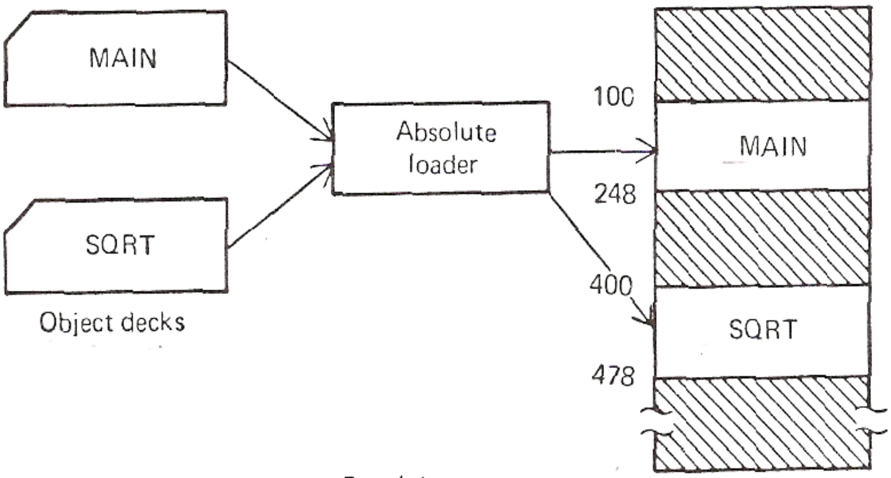


Figure: Example of Absolute Loader

Source program in IBM 360/370 assembly language

Main Program

```
MAIN START      100
        BALR 12,0
        USING       MAIN+2,12
        -
        -
        L           15,ASQRT
        BALR 14,15
        -
        -
ASQRT   DC          F'400'
        END
```

SQRT subroutine

```
SQRT START      400
        USING       *,15
        -
        -
        BR          14
        END
```

Object program for Absolute Loader (from Assembler)

Main program

```
Location      Instruction
100           BALR 12,0
102           -
              -
120           L             15,142(0,12)
124           BALR 14,15
126           -
              -
244           F'400'
248
              SQRT subroutine
400           -
              -
              -
476           BCR           15,14
478           -
```

Design of absolute loader

Recollect that in absolute loader the programmer performs allocation and linking, assembler has absolute address hence the problem of relocation does not arise. The only two functions to be done by absolute loader is

1. To load instructions and data into memory.
2. To transfer control for program execution at the end of loading.

Hence there are only two types of cards used by absolute loader.

1. **Text Card:** This card contains the machine instructions and data created by the assembler along with the address of the memory location where it is to be loaded.
2. **Transfer Card:** It contains the entry point of the program, which is the address where the loader has to transfer control when all instructions are loaded.

FORMAT OF CARDS

Text cards (for instructions and data)

Card	*Contents*

column	
1	*Card type= 0 (for text card identifier)*
2	*Count of no. of bytes of information of card.*
3-5	*Address at which data on card is to be put.*
6-7	*Empty*
8-72	*Instructions and data to be loaded*
73-80	*Card sequence number*

Transfer cards (to hold entry point to program)

Card column	*Contents*
1	*Card type= 1 (for transfer card identifier)*
2	*Count = 0*
3-5	*Address of entry point*
6-7	*Empty*
8-72	*Card sequence number*

Algorithm:

The Absolute loader design uses two data structures

1. CURLOC: It holds address of the current card from column 3-5.
2. LNG: It holds the length of data in the card from column 2 (0 for transfer card)

The algorithm reads the next card from the object deck and sets CURLOC to the address specified in the card.

If card type = 0 then LNG bytes are transferred from card at address CURLOC.

If card type = 1 then control is transferred at address CURLOC.

FLOWCHART

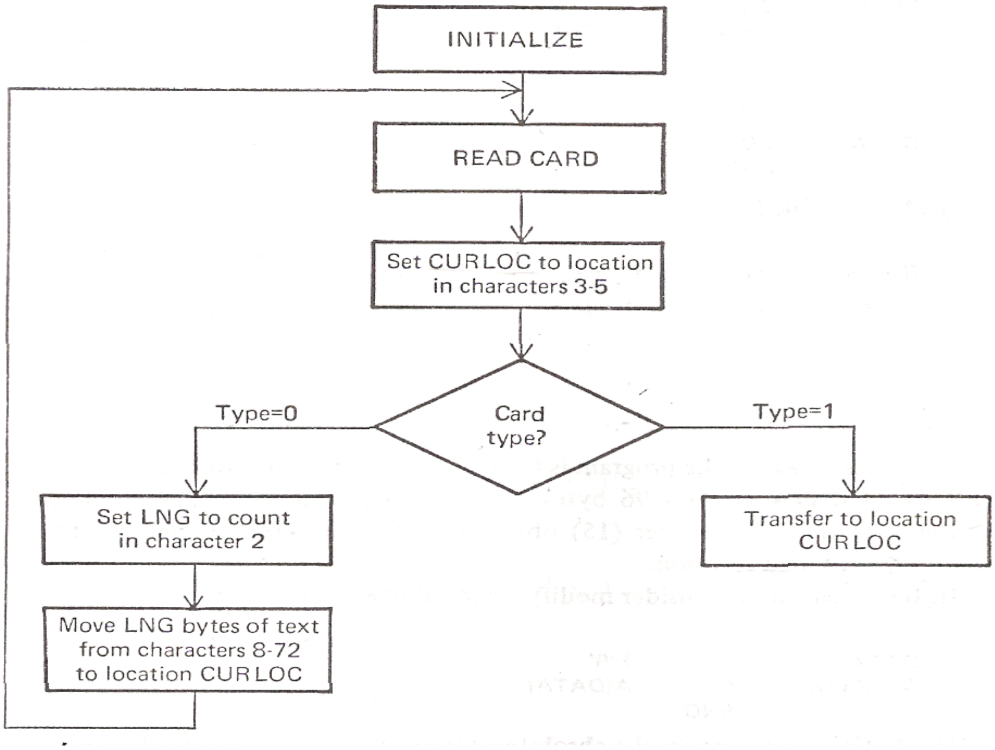

INITIALIZE
READ CARD
Set CURLOC to location in characters 3-5
Card type?
Type=0
Type=1
Set LNG to count in character 2
Transfer to location CURLOC
Move LNG bytes of text from characters 8-72 to location CURLOC

Figure: Absolute loader

2.Relocating loaders

Relocating loaders are used since

- We need not reassemble all subroutines when one is changed.
- Programmer should not perform allocation and linking.

Binary symbolic subroutine (BSS) loader is an example of relocating loader.

Pseudo opcodes used in IBM 360/370 ALP for linking

Pseudo-op EXTRN indicates that symbols defined using EXTRN are defined in other programs but referenced in the present program.

Pseudo-op ENTRY indicates that symbols defined using ENTRY are defined in present program but referenced by other programs.

Operation

1. Assembler assembles each segment independently and passes on to the loader the text and the information of relocation and inter-segment references.
2. The loader uses transfer vector to solve problem of subroutine linkages.

Example: If main program MAIN wishes to transfer to subprogram SQRT then the programmer could write a transfer instruction in MAIN to subroutine SQRT.

3. For each source program the assembler outputs the text prefixed by transfer vector that consists of address containing names of subroutines referenced by the source program.

Example: If SQRT is referenced first then the first location of transfer vector would contain symbolic name SQRT. CALL SQRT would be translated to transfer instruction to the first location in transfer vector.

4. The assembler also provides the loader with additional information such as length of entire program and length of transfer vector portion.
5. After loading the text and transfer vector into the core the loader would also load all the subroutines and would place the corresponding address of the subroutine to each position of transfer vector.
6. Problem of relocation is solved using relocation bits.

Thus all four functions of the loader are all performed by BSS loaders as

1. Allocation: it is performed with the help of program length information.
2. Linking: It is performed with the help of transfer vector.
3. Relocation: It is performed with the help relocation bits.
4. Loading: It is performed in similar way as absolute loader.

Disadvantages

1. Transfer vector linkage is useful only for transfer but not for accessing shared data.
2. Transfer vector increases size of object program in memory.

First disadvantage can be overcome by allowing a common data segment. To implement this, relocation bits are increased to 2 bits.

If 01 then relative to itself (i.e. subroutine or procedure segment)
If 10 then relocated with respect to common data segment
If 00 or 11 then not relocatable (i.e. absolute)

3.Direct Linking Loader (DLL)

DLL has the advantage of allowing the programmer to use multiple procedure segments and multiple data segments and giving the user complete freedom in referencing data or instruction contained in other segments. This provides flexible inter-segment referencing and accessing ability while at the same time allowing independent translation of programs.

The assembler (translator) must give loader the following information with each procedure or data segment -

i) Length of segment and name [SD entry of ESD card]

ii) A list of all symbols in the segment that may be referenced by other segments and their relative location within the segment (LD entry of ESD card)

iii) A list of all symbols not defined in the segment. [ER entry of ESD card]

iv) Information as to where address constants are located in the segment and the description of how to revise their values [RLD entry]

v) The machine code translation of the source program and their relative addresses assigned. [TXT card (text card)]

The assembler provides the above information by producing 4 types of cards in the object deck for each module. They are -*ESD, RLD, TXT and END.* After the end of all modules there is an *EOF/LDT* card

i. ESD card [External symbol dictionary card]

These cards contain information about all symbols that are defined in the program but may be referenced elsewhere and all symbols that are referred in this program but defined elsewhere. There are three types of ESD entry. They are

(*LD: Local definition*): All symbols that are defined in the program but may be referenced elsewhere

(*ER : external reference*): All symbols that are referenced in program but may be defined elsewhere

(*SD - segment definition*): Name of the program and its length

Name	Type	ID	Rel-addr	Length
Y	SD	Y	0	Y
Y	LD	N	Y	N
Y	ER	Y	N	N

Note : Y - entry exists for corresponding type.
N - entry does not exist for corresponding type.

ii. RLD card [Relocation and linkage directory cards]

These cards contain information about those locations in the program that are address sensitive i.e. Value depends upon the address at which program is stored. The entries in RLD card are :

ESD ID	Length (bytes)	Flag + or –	Relative address
ID no. of the symbol from ESD card to which the location is sensitive.		indicates whether value is to be added or subtracted.	Rel-addr of the location which is sensitive with respect to an SD or ER entry

iii. TXT card

These cards contain the actual object code i.e. the translated version of source program with their relative addresses.

iv. END card

This card indicates completion of one subroutine and might contain the starting address of the point of execution if segment is the main program segment.

v. EOF / LDT [End of file or loader terminator card]

This card indicates the end of object deck i.e. end of all the subroutines.

Sample program

```
Src      Load      Rel.
card.     Addr.    Addr.
1       1000        0                    MAIN  START

2                 :                               ENTRY ABC
                  :
3.                :                               EXTRN SQRT, PQR
                  :
|                 :                               :
:                 :                               :

10      1020       20               ABC           ______________

:                 :                               :
:                 :                               :



19      1036      36                     B        DC   A (PQR)
20      1040      40                     PTR      DC   A (SQRT)
21      1044      44                     A        DC   A (ABC)
22      1047      47                             END
```

```
23      1048      0                      SQRT  START

24                                       :        ENTRY PQR

25                                       :        EXTRN ABC

:                        :               :
:                        :               :
:                        :               :
:                        :               :
:                        :               :

30      1076      28                     PQR     _________

:                        :               :
:                        :               :

34      1080      32                     PTR1  DC   A (PQR)

35      1084      36                     PTR2  DC   A (ABC)

36      1087      39                              END
```

Object cards for MAIN routine

ESD card

Src card reference	Name	Type	ID	Relative addr.	Length
1	MAIN	SD	01	0	48
2	ABC	LD	-	20	-
3	SQRT	ER	02	-	-
3	PQR	ER	03	-	-

RLD card

Src card Reference	ESD ID	Length (bytes)	Flag + or -	Relative addr.
19	03	4	+	36
20	02	4	+	40
21	01	4	+	44

Object cards for SQRT routine

ESD card

Src card Reference	Name	Type	ID	Relative addr.	Length
23	SQRT	SD	01	0	40
24	PQR	LD	-	28	-
25	ABC	ER	02	-	-

RLD card

Src card reference	ESD ID	Length (bytes)	Flag + or -	Relative addr.
34	01	4	+	32
35	02	4	+	36

Design of two pass direct linking loader

Since the direct-linking loader may encounter external references in an object deck which cannot be evaluated until a later object deck is processed, this type of loader requires two passes. Their functions are very similar to those of the two pass assembler.

Functions of two passes of two pass DLL

Pass I: The major function of pass1 of a DLL is to allocate and assign each program a location in core and create a symbol table filling in the values of external symbols.

Pass II: The major function of pass2 is to load the actual program text and perform the relocation modification of any address constants needing to be altered.

Databases needed for two pass DLL

Pass I: Databases needed for the first pass are

1. Input object file: The object deck produced by assembler having 4 types of cards.
2. The Initial Program Load Address (IPLA): It is supplied by programmer or the operating system that specifies the address to load the first segment.
3. A Program Load Address (PLA): It is used to keep track of each segments assigned location.
4. The Global External Symbol Table (GEST): The GEST is used to store the external symbols defined by means of a segment definition (SD) or local definition (LD) entry or an External symbol dictionary (ESD) card.

When these symbols are encountered during pass1, they are assigned an absolute core address; this address is stored along with the symbol in the GEST.

External Symbol	Assigned core address

5. A copy of input to be produced for pass2.

Pass II: Databases needed by Pass2 are

1. Copy of object program deck as produced by pass1 or the object deck.
2. The Initial Program Load Address (IPLA) : Same as Pass1.
3. The Program Load Address (PLA): Same as Pass1.
4. The Global External Symbol Table (GEST) produced by pass1.
5. An array Local external symbol array (LESA):The external symbol to be used for relocation or linking is identified on the RLD cards by means of an ID number rather than the symbol’s name.

This ID number must match an SD or ER entry on the ESD cards. This technique both saves space on the RLD cards and speeds processing by eliminating many searches of GEST. In pass 2 of the loader the GEST and ESD information are merged to produce the local external symbol array (LESA) that directly related ID number and value.

It is necessary to create a separate LESA for each segment, but since LESA arrays are produced one at a time, the same array can be revised for each segment.

ESD ID ***(Index)***	**Assigned core address of corresponding symbol** ***(4-bytes per entry)***
1	1000
2	1048
-	
255	

255 entries maximum

Databases for the example is

PASS1 of DLL –

GEST: –

External Symbol	**Assigned core address**
MAIN	1000
ABC	1020
SQRT	1048
PQR	1076

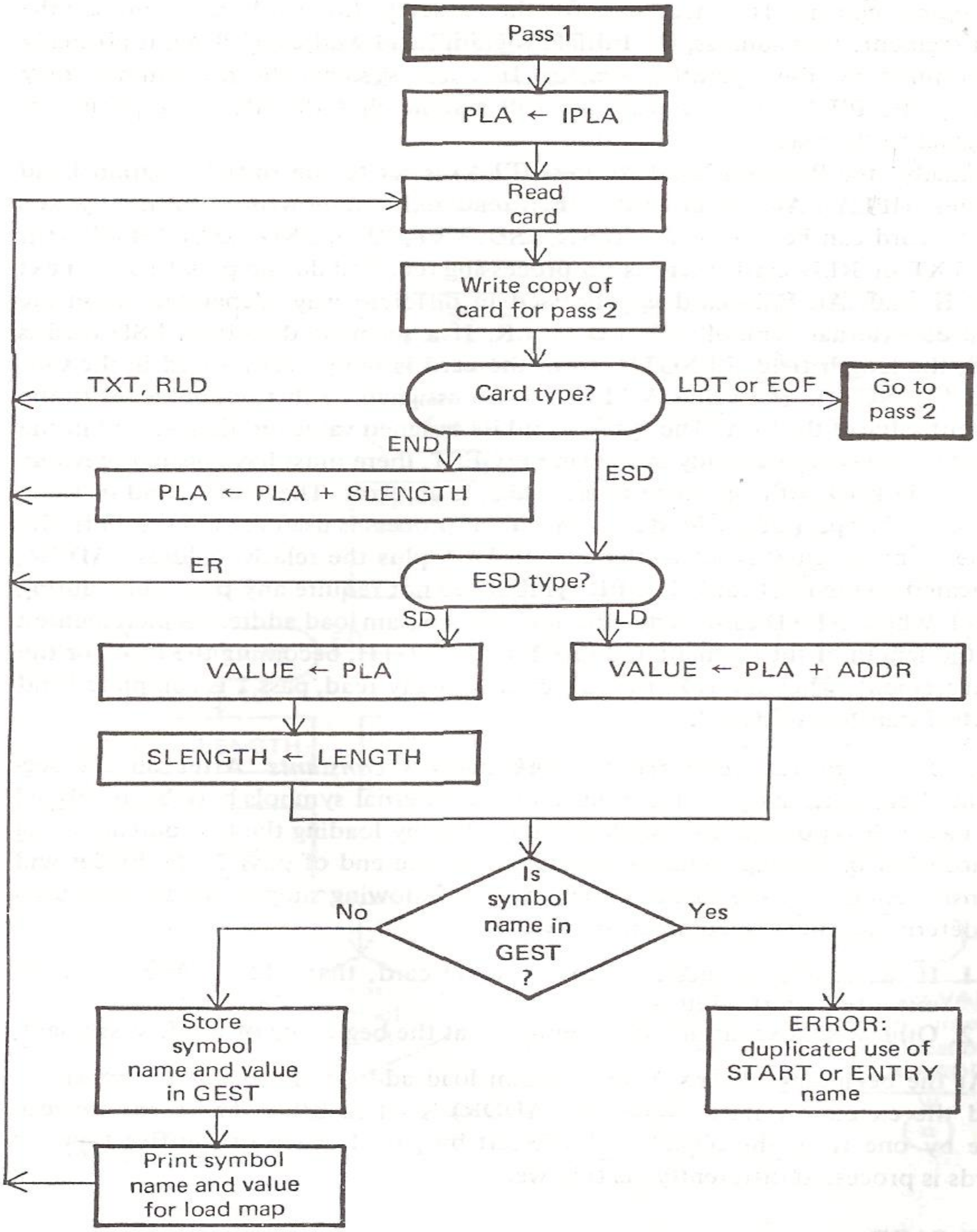
PASS2 of DLL –

LESA for MAIN

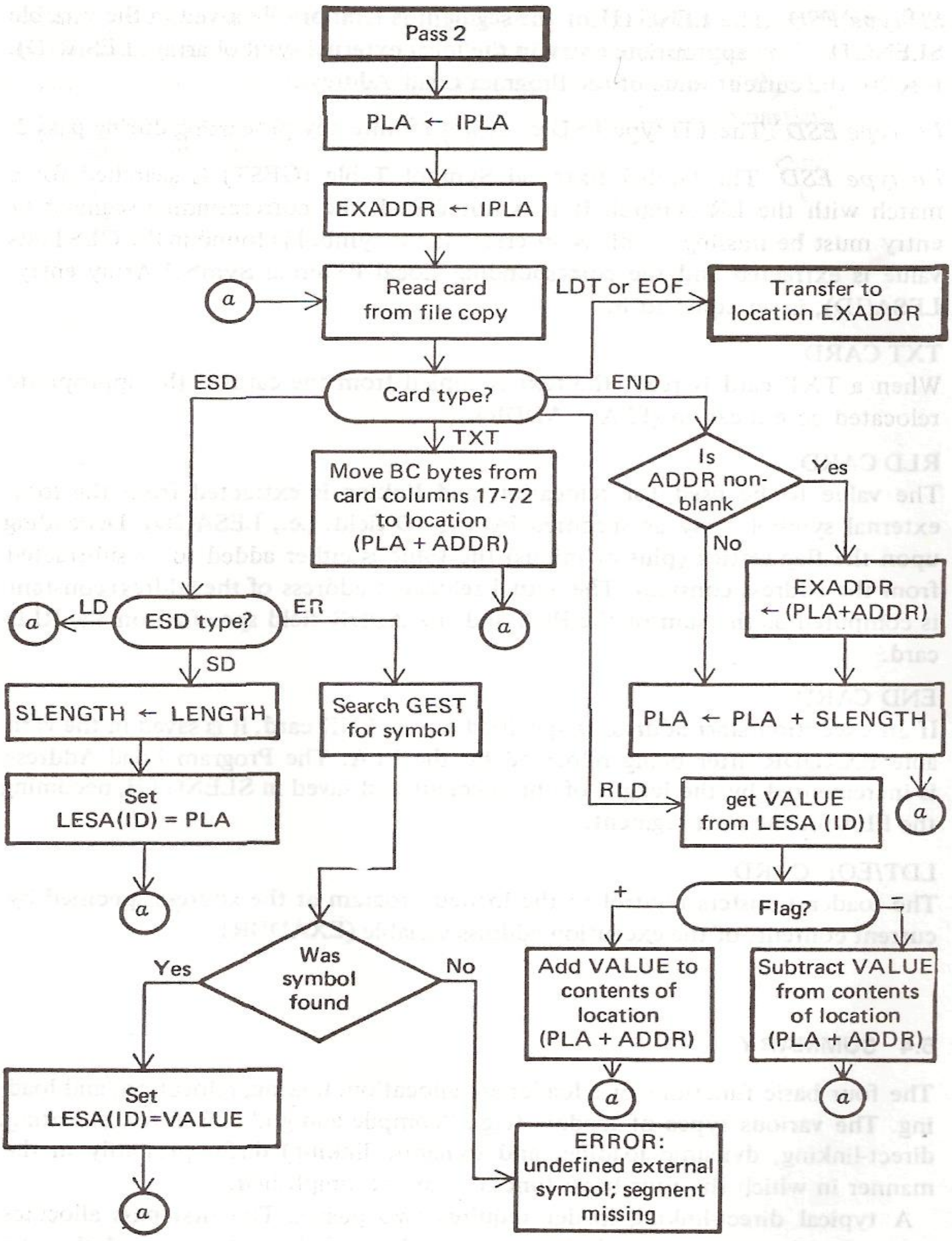
ESD ID	**Assigned core address**
01	1000
02	1048
03	1076

LESA for SQRT

ESD ID	**Assigned core address**
01	1048
02	1020



Pass I Flowchart



Pass II Flowchart:

BINDERS AND MODULE LOADERS

Direct linking loaders perform all four functions hence size is large and hence wastage of memory. Every time the program executes linking needs to be redone. Hence the functions of the loader can be split up into a BINDER and MODULE LOADER.

Hence complex task of linking would be performed by the BINDER and stored on secondary media.

When the program is ready for execution the MODULE LOADER would load the program into main memory. The module loader is considered to be smaller than Direct linking loader.

Binders are of two types.

1. Core image builder:

* It performs allocation, relocation and linking.

* It builds an output similar to actual object file as would be loaded in memory.

* Allocation address cannot be modified by the module loader and hence absolute loader can be used.

DISADVANTAGE

Every time program has to be loaded in same memory location.

2. Linkage editor:

- It performs the complex function of linking.
- It passes linked modules for allocation and relocation by the module loader.
- Relocatable loader can be used which performs allocation, relocation and loading.
- Need of linking arises when program modules call each other or share some common data.
- Only references to external data or programs are handled by linkage editor.
- References to internal programs and local data would have to be handled by translator itself.
- Assembler flags all external symbolic references for processing by the linkage editor.
- Linkage editor builds LINK-TAB and RELOC-TAB for every program.
  - Assembler makes an entry for external references like CALL instruction in LINK-TAB.
  - Assembler makes entry for address sensitive statements in RELOC-TAB.
- Input to the linkage editor is a set of object modules which are to be linked and load origin address.

Algorithm:

I. Repeat steps II and III for all object modules.

II. Assign linked address to all programs in object module.

III. Build a NAME TABLE storing program names, linked origin address and relocation factor.

IV. Perform steps V and VI for each program module.

V. Perform relocation for each module using the information from RELOC-TAB and relocation factor for NAME TABLE.

VI. Perform linking for each module using LINK-TAB and NAME TABLE.

Algorithm performs linking in two passes –

Pass I: During first pass load address is assigned to each module, relocation factor is calculated and an entry is made in NAME TABLE with the load address.

Pass II: During second pass module is processed for relocation and linking.

Since load address is available in NAME TABLE linking is simply a matter of taking up the appropriate load address and assigning it to the bytes and relocation is a matter of updating address-sensitive instructions specified in RELOC-TAB by relocation factor in NAME TABLE.

MS DOS LINKER:

MSDOS compilers and assemblers produce object modules(.OBJ) which contains a binary image of the translated instructions and data of the program.

MS-DOS LINK is a linkage editor that combines one or more object modules to produce an executable machine language program (.EXE).

Record types:

THEADR: Translator header specifies name of object module

External symbols and references

PUBDEF: Public names like symbols defined using ENTRY.

EXTDEF: External references like symbols defined using EXTRN.

TYPEDEF: Defines the type for symbols in PUBDEF AND EXTDEF.

Segment definition and grouping

LNAMES: It contains list of all the segment & class names that could be referred by subsequent SEGDEF records.
SEGDEF: It describe segments in object module including name, length & alignment. It also gives position in LNAMES.
GRPDEF: It specify how the segments are combined into groups it refers to segment in LNAMES.

Translated instructions and data

LEDATA: It contains translated instructions & data
LIDATA: It specify translated instructions & data that occur in a repeating pattern.

FIXUPP: Relocation & linking information

MODEND :End of object module. MODEND record marks end of object module and contain a reference to the entry point of program

MSDOS Linker performs its processing in 2 passes.

PASS 1:

1. It Computes starting address for each segment in a program. Generally segments are places into the executable program in the same order that the SEGDEF records are processed. Segments with same name and class from different object modules are combined. Segments with same class but different names are concatenated. The starting address initially associated with a segment is updated during PASS 1 as these concatenations & combination are performed.
2. PASS 1 constructs a symbol table that associates an address with each segment (using the LNAMES, SEGDEF & GRPDEF records) and each external symbol (using EXTDEF & PUBDEF records). If unresolved external symbols remain after all object modules have been processed, LINK searches the specified libraries.

PASS 2:

1. LINK extracts the translated instructions and data from the object modules and builds and image of the executable program in memory.

It there is not enough memory available to contain entire executable image, LINK uses temporary disk file in addition to all of the available memory.

2. LINK process each LEDATA and LIDATA record along with the corresponding FIXUPP record It places binary data from LEDATA and LIDATA records into the memory image at locations that reflect segment addresses computed during PASS I.
3. Relocations within a segment (caused by combining or grouping segments) are performed and external references are resolved.

Relocation operations that require starting address of a segment are added to a table of segment fixupps.

This table is used to perform relocations that reflect the actual segment addresses when the program is loaded for execution.

4. After memory image is complete LINK writes it to the executable (.EXE) file. This file includes a header that contains the table of segment fixupps, info about memory requirements and entry points and initial contents of registers CS and SP.

Dynamic loading and Dynamic linking:

Dynamic Loading:

In dynamic loading, the subroutines are loaded into core at different time.

In case of rest of the loader scheme, all the subroutines are loaded into core at the same time.

There is trouble occur when the total amount of core required by all these subroutines exceeds the amount available.

There are several hardware techniques, such as paging and segmentation, that are used to solve the above mentioned problem. .

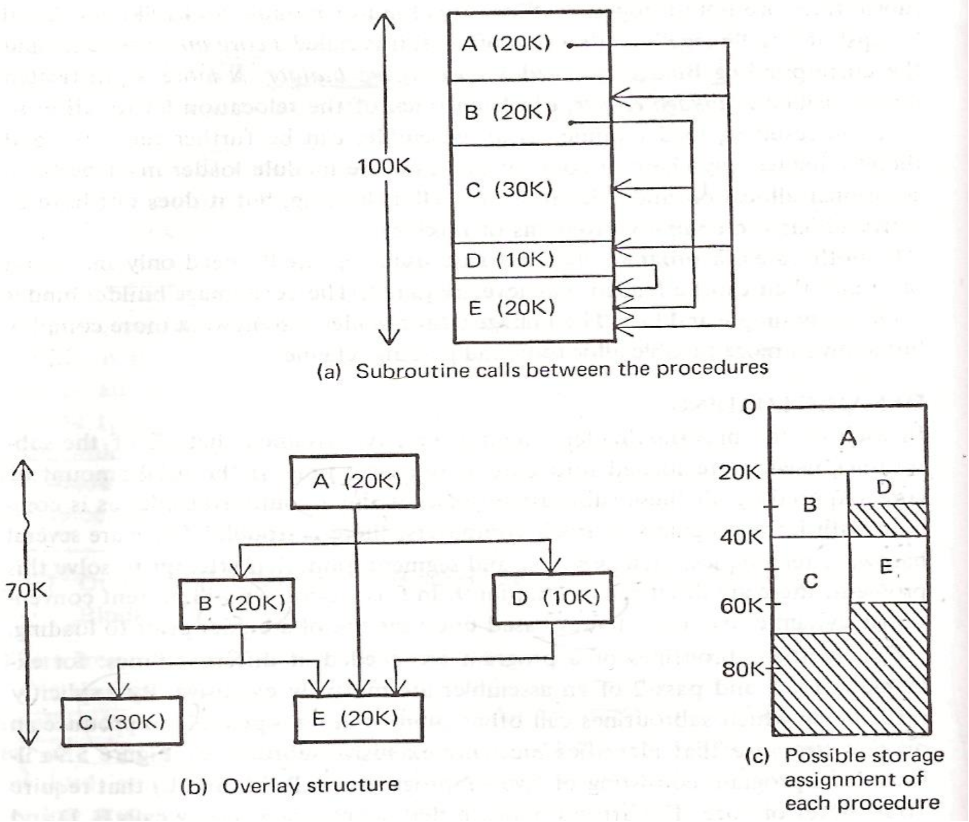
Dynamic loading schemes based upon the use of a binder prior to loading can be used to solve the same problem:

The subroutines of a program are needed at different times.

Example:

Pass 1 and Pass 2 of an assembler are mutually exclusive. By recognizing which subroutine calls other subroutines, it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

The following figure illustrates a program consisting of five subprograms (A, B, C, D, E).



(a) Subroutine calls between the procedures

(b) Overlay structure

(c) Possible storage assignment of each procedure

The subprograms require 100 KB of core. The arrow indicates that subprogram A only calls B, D & E.

Subprogram B only calls C & E,

Subprogram D only calls E,

Subprogram C & E do not call any other routines.

The figure(b) highlights the interdependencies between the procedures.

For the overlay structure to work properly, it is necessary for the module loader to load the various procedures as they are needed.

There are many binders capable of processing and allocating an overlay structure.

The portion of the loader that actually intercepts the "calls" and load the necessary procedure is called the overlay supervisor or simply the flipper.

The overall scheme is called dynamic loading or load-on-call (LOCAL).

Dynamic Linking :

A major disadvantage of all of the previous loading schemes is that if a routine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.

e. g. If the programmer has placed a call statement in this program but his statement was never executed because of a condition that branched around it.

All these schemes require the programmer to explicitly name all procedures that might be called.

In dynamic linking mechanism, loading and linking of external references are postponed until execution time. The assembler produces text, binding and relocation information from a source language deck.

The loader loads only the main program.

If the main program should execute a transfer instruction to an external address or should reference an external variable, the loader is called, only then is the segment containing the external reference loaded.

Advantage –

There is no overhead incurred unless the procedure to be called or referenced is actually used.

Disadvantage -

There is complexity incurred since we have postpones most of the binding process until execution time.

Who loads the loader?

Bootstrap Loaders:

Given an idle computer with no program in memory, how do the things start?

In this situation, with the machine empty and idle, there is no need for program relocation. We can simply specify the absolute address for whatever program is loaded first. Most often, this program will be operating system, which occupies a predefined location in memory. This means we need some means of accomplishing the functions of an absolute loader.

1. On some computers, an absolute loader program is permanently resident in ROM. When some hardware signal occurs (for example the operator pressing a "system start" switch) the machine begins to execute this ROM program.

2. On some computers, the program is executed directly in the ROM and on others the program is copied from ROM to main memory and executed there. Disadvantage of this scheme is that ROM need to be changed if modifications.

3. An intermediate solution is to have ROM that reads a fixed length record from some device(floppy diskettes / Compact Disks / Hard Disks) into memory at a fixed location. After the read operation is complete, control is transferred to the address in memory where the record was stored. This record contains machine instructions that load

the absolute program that follows. If the loading process requires more instructions than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of still more records-hence the term bootstrap. The first record is generally referred as ***bootstrap.*** Such record is added to the beginning of all object program (operating systems) that are to be loaded into an empty idle system.

University questions on this topic:

1. Explain the design of Absolute loader. **(10M)**

CMPN May 04, May 05, May 06, May 07, May 08, Dec 08

2. Explain the design of direct linking loader. **(10M)**

CMPN Dec 04, May 06, Dec 06, May 07, Dec 07 ,May 08, Dec 08 and IT Dec 07

3. Describe the working of a direct linking loader. Explain in detail the various data structure used.

IT May 04(10M), Dec 05 (10M), Dec 04 (7 M)

4. Explain the relocation and linking concepts.

CMPN May 04 (10M)

5. Explain linkage editor.

CMPN May 04, May 05 (6M)

6. Explain linkage editor and dynamic binding.

CMPN May 04 (10M)

7. Write short note on MSDOS linker.

CMPN May 08 (5M)

8. What is binding ? Explain static and dynamic binding.

CMPN May 04 (10M), CMPN May 05 (04M), CMPN May 06 (06M), CMPN Dec 06 (06M)

9. What is the need of linking editor in systems programming? Explain its working in brief.

CMPN Dec 04 and IT May 07, Dec 07 (10M)

10. What are basic functions of Loader ?

CMPN May 05 (04M), May 10(5M)

11. Explain relocatable loader with reference to following examples. Calculate relocatable address for following programs. Consider all instructions as 1-byte instruction and program is stored from location 0000H and it is loaded starting at location 2010.

```
LOAD      1234
ADD       9000
BRANCH    4567R
STORE     7000R
```

Where R indicate relocatable address.

IT May 06 (10M)

12. What is loader? Explain the loader functionality and its components in detail.
CMPN Dec 05 (10M)

13. What is the need of linkage editor in systems programming? **CMPN Dec 06 (04M)**

14. What is Loader? Explain the four basic functions of a loader. Also differentiate absolute loader vs. Relocating loader stating who performs which function in the particular loading scheme.
IT Dec 04 (10M)

15. What is Loader? Explain the four basic functions of a loader. Also differentiate absolute loader vs. Relocating loader stating who performs which function in the particular loading scheme.
IT Dec 04 (5M)

16. Design of direct linking loader.
IT Dec 04 (07M)

17. Dynamic linking.
IT Dec 04 (8M), Dec 05 (07M), Dec 07 (5M)

18. Dynamic loading and linking
CMPN Dec 07, May 08 (10M)

19. What is relocating loader ? Explain who performs the **four** functions of this loader.
IT May 05 (06M)

20. Explain the terms BSS, overlay and binding with respect, to loaders.
IT May 05 (06M)

21. Differentiate between linkage editor and linking loader.
CMPN May 06 (8M), Dec 07 (10M), May 10 (5M)

Chapter 5

Compiler

Compilers.

Definition: A compiler is a program that takes as input a source program written in high level language and produces as output an equivalent sequence of assembly or machine level instructions.

Phases of a compiler

The process of compiler is so complex that it is not reasonable to consider the compilation process as occurring in one single step. For this reason it is partitioned into a series of subprocesses called phases. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

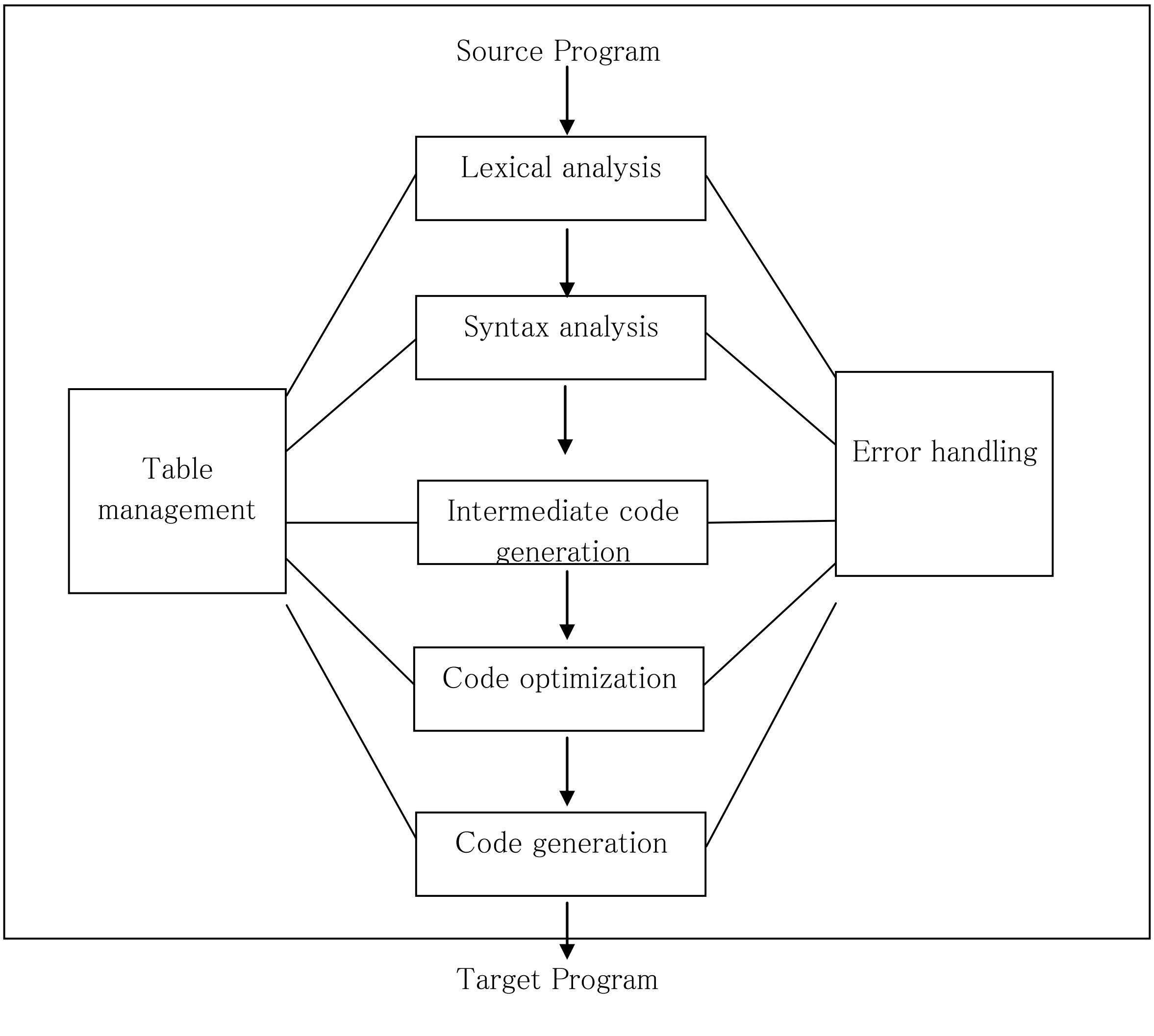
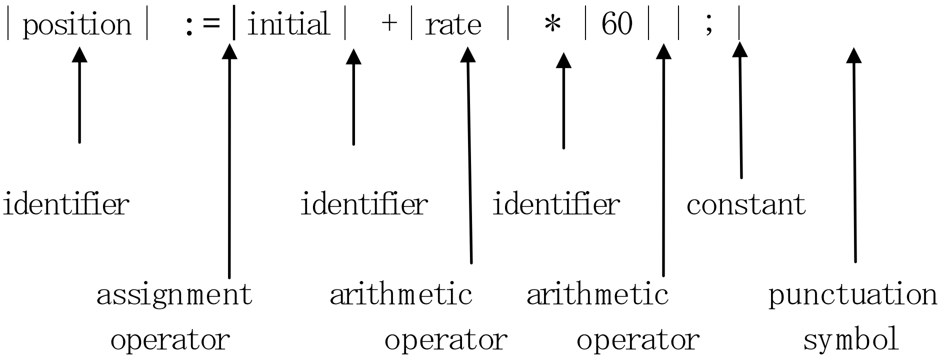


Figure: Phases of a Compiler

1. Lexical analysis

The lexical analyzer /scanner separates characters of the source language into groups that logically belong together. These groups are called tokens. The blanks separating the characters of tokens would be eliminated during lexical analysis. Examples of types of tokens are keywords, operators, punctuation symbols, identifiers and constants.

Example:



2. Syntax analysis

The syntax analyzer groups tokens together into syntactic structures. These syntactic structures of the source program are represented by parse tree. The leaves of this parse tree are the tokens generated by scanner. The interior nodes are strings of tokens that logically belong together.

Example:

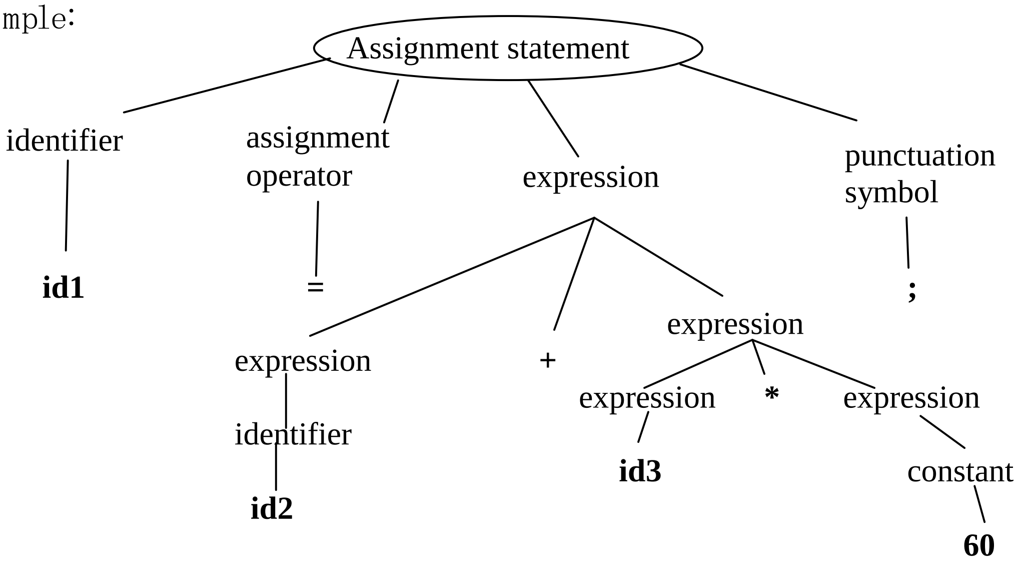


Figure: Parse tree

3.Intermediate code generation

A notational framework for intermediate code generation that is an extension of context free grammars is called syntax-directed translation scheme. It allows semantic actions/rules to be attached to the productions of context-free grammar. Semantic analysis performs type checking. The compiler checks that each operator has operands that are permitted by the source language specification.

Many compilers convert source to an intermediate level language called intermediate code instead of assembly or machine. Doing so makes optimization easy. Intermediate code generator uses the structure produced by the syntax analyzer to create a stream of simple instructions.

One type of intermediate form is "three - address code ". Three address code consists of a sequence of instructions each of which has at most three operands.

```
Example      t1   = 60
             t2  = id3 * t1
             t3  = id2 + t2
             id1     = t3
```

4.Code optimization

The code optimization phase attempts to improve the intermediate code so that faster running machine code will result. Its output is another intermediate code that does the same job but in a way that saves time and/or space.

```
Example   t1 = id3  * 60
            id1   = t1 + id2
```

5..Code generation

The final phase of the compiler is the generation of target code. This phase decides on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done.

```
 Example
      MOV          R1, A(id3)
      MUL          R1, 3CH
      ADD          R1,  A(id2)
      MOV          id1, R1
```

Symbol table management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifiers. These attributes may provide information about the storage allocated for an identifier, its type, its scope, etc. A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier.

Data structures for symbol table

For creating a symbol table we can use anyone of the following mechanisms.

(1) Linear List

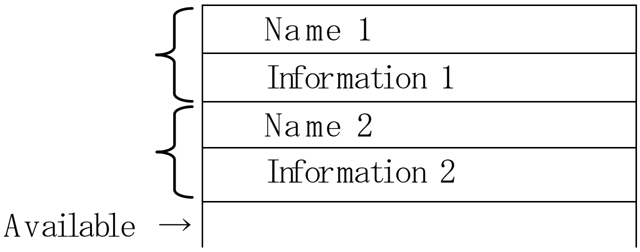
(2) Search Tree

(3) Hash Table

(1) Linear List:

The simplest and easiest to implement data structure for a symbol table is the linear list of records. It uses a single array/list to store names and their associative information. New names are added to the list in the order in which they encountered in the program. "Available" pointer/index indicates the beginning of the empty position of the array/list.

Example



Drawback:

It is inefficient because the table should be searched several times during compilation and searching process will be very slow for a linear list.

Advantages:

a) Easy to implement.
b) Requires less space.

2) Search Tree:

A more efficient approach to symbol table organization is to add two link fields left and right to each record, we use these fields to link the records into a binary search tree. This tree has the property that all names NAMEj accessible from NAMEi by following the link LEFTi, will precede NAMEi in the alphabetical order (i.e. NAMEj < NAME i). Similarly all the names coming in the right sub-tree will succeed NAMEi in the alphabetical order (i.e NAMEj > NAMEi). In this technique searching time is reduced as compared to linear list.

3. Hash Table:
Two tables called a hash table and a storage table are used for creating a symbol table. The hash table consist of K locations numbered 0, 1, 2,... K-1. These locations are pointers to the storage table. They point to the heads of K separate linked lists. Each record in the symbol table appears on exactly one of these lists. To search for a name in the symbol table, we apply to the name a hash function h (i.e h(name)) which returns an integer i between 0 and K-1. It is on the list numbered "i" that the record of name belongs.

Error handling

One of the most important functions of a compiler is the detection and reporting of errors in the source program. Error can be encountered by all phases of a compiler.

Example of some errors detected by various phases of compiler:

1. The lexical analyzer may be unable to proceed because the next token in the source program is misspelled.

2. The syntax analyzer may be unable to infer a structure for its input because a syntactic error such as missing parenthesis has occurred.

3. The ICG may detect an operator whose operands have incompatible types.

4. The code optimizes may detect that certain statements can never be reached.

5. The code generator may find a compiler - created constant that is too large to fit in a word of the target machine.

6. While entering information into the symbol table the book-keeping routine may discover an identifier that has been multiply declared with contradictory attributes.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the compiler must take corrective actions such as to modify the input with defaults so that the phase detecting the error and subsequent phases can continue processing and looking for subsequent errors.

Comparison of Compiler and Interpreter

	Compiler	Interpreter
(1)	It translates the whole program written in HLL into its equivalent machine code in one go.	It translates line by line the source program written in HLL into its equivalent machine code.
(2)	Speed of execution is fast as it saves a lot of time when the program has to be executed repeatedly.	The speed of execution is slow as it re-translates the source when the program has to be executed repeatedly.
(3)	It requires large memory space as the code size of the compiler is larger.	It requires less memory space as the code size of the interpreter is smaller.
(4)	It creates an object file containing the translated program.	It does not create an object file but loads the translated program into memory for execution.
(5)	Software cost is high.	Software cost is less.
(6)	Example C compiler, JAVA Compiler.	Example BASIC interpreter, JAVA Interpreter in JVM.

University questions

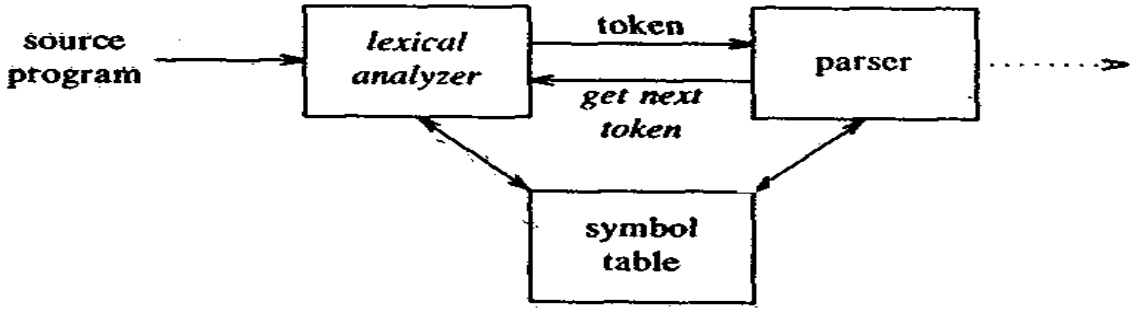
1) What is Compiler? Draw and explain the block diagram for the structure of general compiler.
   IT Dec04 (10M)
2) Explain the phases of compilers with respect to the following statement
   position := initial + rate * 60
   CMPN Dec04 (10M),Dec06(10M)
3) Explain the phases of compilers with suitable example
   CMPN May04 (10M),Dec08 (10M),May10 (10M)
4) What are the various phases of compiler ? Explain.
   CMPN Dec05 (10M)
5) "In every compiler there is an integrated assembler." Justify.
   IT May05 (06M)

Chapter 6

Lexical analysis

Lexical analysis

The lexical analyzer is the first phase of a compiler. Its main function is to read the input source program character by character and produce as output a sequence of primitive units called tokens that the parser uses for syntax analysis. This interaction, summarized schematically in figure below.



Interaction of lexical analyzer with parser

Keywords, identifiers, constants and operators are examples of tokens. Although the task of lexical analyzer/ scanner could be a separate pass, that is to convert the entire source program into a sequence of tokens and output it on an intermediate file. But scanner will rarely do this in separate pass, more commonly **scanner and parser are in a single pass.** The lexical analyzer is a subroutine of the parser i.e. scanner will operate under the control of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Since the lexical analyzer is the only phase of the compiler that reads the source text, it may also perform certain **secondary tasks** at the user interface. One such task is stripping out from the source program comments and white; space in the form of blank, tab, and newline characters. Another is correlating error messages from the compiler with the source program in terms of line numbers. It also stores information regarding symbols in symbol table, it can also perform case conversion if the language is not case sensitive so as to have uniform case for all symbols.

The **main reason for separating the analysis phase of compiling into lexical analysis and syntax analysis** is simpler design. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases. For example, a parser that takes care of comments and white space is significantly more complex than one that can assume comments and white space have already been removed by a lexical analyzer. This also improves efficiency of compiler.

Role of Finite State Automata and RE in scanner design

Tokens, Patterns, Lexemes

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a **pattern** associated with the token. Regular expressions are advantageous in defining such patterns, and hence they can act as recognizer for the tokens. The pattern i.e regular expression is said to match each string in the set.

A **lexeme** is a sequence of characters in the source program that is matched by the pattern for a token.

For example, in the Pascal statement const pi = 3.1416;
The substring pi is a lexeme for the token "identifier" having pattern rule specified using regular expression as letter(letter|digit).*

Finite Automata

While constructing the lexical analyzer we first design the regular-expression for the token. A method to convert regular-expression into its recognizer is a LEX process which constructs a generalized transition diagram for that expression called Finite Automata. The Finite Automata is the recognizer for a language L that takes as input a string w and answers yes if w is a sentence of L and no otherwise.

Design of Lexical-analyzer

In this section, we consider the design of a software tool that automatically constructs a lexical analyzer from program in the Lex language. Although we discuss several methods, and none is precisely identical to that used by the UNIX system Lex command, we refer to these programs for constructing lexical analyzers as Lex compilers.

We assume that we have a specification of a lexical analyzer of the form

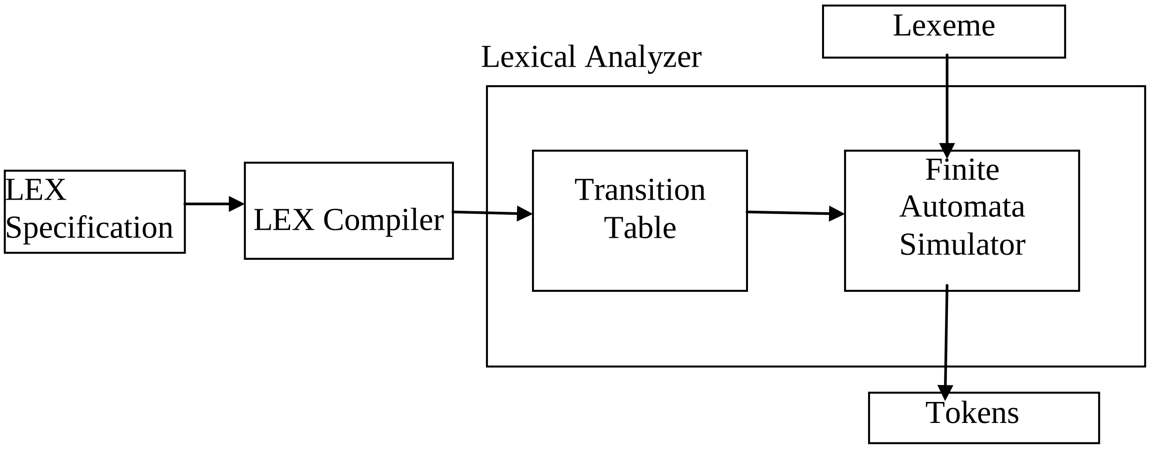
$$\begin{array}{ll} p_1 & \{action_1\} \\ p_2 & \{action_2\} \\ \ldots & \quad\ldots \\ p_n & \{action_n\} \end{array}$$

where each pattern p_i is a regular expression and each action $action_1$ is a program fragment that is to be executed whenever a lexeme matched by p1, is found in the input.

Our problem is to construct a recognizer that looks for lexemes in the input buffer. If more than one pattern matches, the recognizer is to choose the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the first-listed matching pattern is chosen.

A finite automaton is a natural model to build a lexical analyzer, and the one constructed by our Lex compiler has the form shown in figure below. The Lex compiler constructs a transition table for a finite automaton from the regular expression patterns in the Lex specification. The lexical analyzer itself consists of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer.

The implementation of a Lex compiler can be based on either nondeterministic or deterministic automata. The transition table of an NFA for a regular expression pattern can be considerably smaller than that of a DFA, but the DFA has the advantage of being able to recognize patterns faster than the NFA.



University questions

1) Consider the regular-expression r = (a/b)*abb. Construct the NFA for this expression and convert this NFA to minimized DFA.
   CMPN May06 (10M)
2) For the regular-expression (0+1)*01. Construct a NFA for this expression and convert this NFA to DFA.
   CMPN Dec05 (10M)
3) What is regular-expression? Give regular-expression which denotes all the strings whose length is not 2. Draw the NFA for the same.
   CMPN May05 (4M), IT Dec04 (10M)
4) Define Finite State Automata. What is their role in compiler theory? Explain in detail.
   CMPN Dec04 (10M),Dec06 (04M),May10 (05M)
5) Explain the role of Finite State Automata and regular expressions in compiler design.
   CMPN May07 (10M)

Chapter 7

Syntax analysis

Syntax Analysis

A syntax analyzer takes as input a stream of tokens and generates a parse tree. Its task is to analyze the syntactic structure of the program and check for errors. To check the syntax we need to provide to it the syntax of the programming language.

Overview of types of grammar (Chomsky Hierarchy)

Grammar

A grammar is denoted using 4 components G = (V, T, P, S)
where
V is finite set of variables/ non-terminals/ non-leaf nodes
T is finite set of terminals/ leaf nodes
P is finite set of productions. Each production is of type $\alpha \rightarrow \beta$
S is the start symbol

Types

Grammars are classified on the basis of the productions used in them.
Chomsky suggested four types:

Type – 0 Unrestricted Grammar:

As name suggests these grammars have no restriction on the productions . All productions can be of the form $\alpha \rightarrow \beta$ with $\alpha \neq \epsilon$.
where both α and β are strings of terminals and non-terminals. Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are not relevant to specification of programming languages.

Type – 1 Context Sensitive Grammar:

These grammars are known as context sensitive grammar because their productions specify that derivation or reduction of strings that can take place only in specific contexts. Each production in Type – 1 has the form

$$\alpha A \beta \longrightarrow \alpha \theta \beta$$

Thus, a non-terminal 'A' can be replaced by 'θ' only when it is enclosed by the context α preceding it and β following it.
These grammars are particularly used in semantic specification since recognition of many semantic structures is context sensitive in nature. Semantic specifications can also be done by defining an extension to CFG(type 2) as discussed in next chapter.

Type - 2 Context Free Grammars:

These grammars impose no context requirements on derivations or reductions. A typical Type - 2 production is of the form.

$$A \longrightarrow \alpha$$

which can be applied independent of its context. These grammars are therefore known as context free grammars. CFG are ideally suited for syntactic rule specification of programming language.

Type - 3 Regular Grammars :

Type - 3 grammars are characterized by productions of the form

$$A \longrightarrow tB \mid t \text{ or}$$
$$A \longrightarrow Bt \mid t$$

The specific form of the RHS alternatives – namely a single terminal or a string of terminals followed by/ preceded by a single NT–gives some practical advantages in scanning.

However the nature of productions restricts the expensive power at these grammars. Example nesting of constructs or matching of parenthesis cannot be specified using such productions. Hence the use of Type - 3 productions is restricted to specification of lexical units. e.g. identifiers, constants, labels etc.

Type - 3 grammars are also known as *linear grammars or regular grammars.*

These are further categorized into *left - linear* and *right - linear* grammars depending on whether the NT in the RHS alternative appears at the extreme left or extreme right.

A parser/ syntax analyzer to check for syntax errors in the program are based on Type2 that is Context–free grammar.

Overview of Context free grammars –

The syntax of programming language constructs can be described by context - free grammars or BNF (Backus - Naur form) notation. A constant free grammar consists of terminals, non–terminals, a start symbol and productions. Every production has a single variable on left and a sequence of terminals and variables on right .

Terminals

Terminals are basic symbols from which strings are formed. They are the tokens generated by scanner.

Non-terminals -

Non-terminals are syntactic variables that denote sets of strings as their productions.

Example "if E then S1, else S2 "is a statement

stmt **if** ~~expr~~ **then** stmt **else** stmt.

'stmt' and 'expr' are non-terminals

The non-terminals defines set of strings that help define the language generated by the grammar. In a grammar, one non terminal is distinguished as the start symbol. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. A language that can be generated by a context-free grammar is said to be a context-free language.

If two grammars generate the same language, the grammars are said to be equivalent.

If S derives L in multiple steps, where L may contain non-terminals, then we say that L is a sentential form of G.

A sentence is sentential form with no non-terminals.

Derivations in which only the leftmost non-terminal in any sentential form is replaced at each step. Such derivations are termed leftmost. Similar definitions hold for right sentential form.

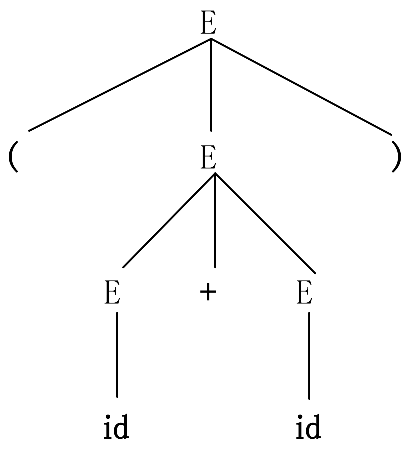
Parse trees / Derivation trees -

Parse tree is a graphical representation for a derivation.

Example: The grammar is

E -> E + E |E * E | (E) | -E| id

Then the parse tree for (id + id) will be



Ambiguity

An ambiguous grammar is one that has more than one derivations for a sentence that belongs to the language that is it produces more than one leftmost or more than one rightmost derivation for the same sentence.

Example:

For the above given grammar the sentence id + id * id has two distinct leftmost derivations. Hence the grammar is ambiguous.

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

Eliminating ambiguity

Consider the grammar,

E -> E + E | E - E |
E * E | E / E |
E ˆ E | (E) |
- E | id

This is an ambiguous grammar.

To eliminate this ambiguity, we first find out the precedence.

The following precedences are in decreasing order:

- (Unary minus) ↓ *Highest*

ˆ

* , /

+, - *Lowest*

(Higher precedence operators lie at the leaves and lower precedence operators at the root.)

Hence unambiguous grammar will be -

expression -> expression + term | expression – term | term

term -> term * factor| term / factor | factor

factor -> primary ˆ factor | primary

primary -> – primary | element

element -> (expression) | id

Parser

A parser for grammar G is a program that takes as input a string 'w' and produces as output either a parse tree for 'w' , if 'w' is a sentence of G, or an error message indicating that 'w' is not a sentence of G.

The two basic types of parsers for content free grammars are ***bottom up*** and ***top down***

Bottom-up parsers build their parse trees from the bottom (leaves) to the top (root)

Top-down parsers start with the root and work down to the leaves.

Top-Down Parsing

The parser of this type read the input stream of tokens (terminals) from scanner in left to right order (L) and construct the parse tree from top to bottom based on leftmost derivation (L). Hence such parsers are called LL parsers. In general they are LL(k) where 'k' represents number of tokens read ahead of time by the parser. We study later LL(0) and LL(1) type design technique. Since LL are based on leftmost derivation so they may lead to infinite loop if the grammar has left recursion. Hence for top-down parser left recursion must be eliminated.

Left Recursion

[Informally, If the leftmost non-terminal of the right hand side of a production is same as the L.H.S of the production it causes left recursion]

A grammar is left recursive if it has a non-terminal "A" such that there is a derivation of the form

$$A \rightarrow A\alpha \ / \ \beta$$

Top down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed.

Elimination of left recursion

$$A \rightarrow A\alpha \;/\; \beta$$

could be replaced by non-recursive productions.

$$A \rightarrow \beta\, A'$$

$$A' \rightarrow \alpha\, A' / \epsilon.$$

Without changing the set of strings derivable from A.

For e.g. refer class notes

Left factoring –

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the A-production to defer the decision until we have seen enough of the input to make the right choice.

If we have two productions

stmt -> if expr then stmt else stmt | if expr then stmt

On seeing the input token 'if', we cannot immediately tell which production to choose to expand 'stmt'.

Example:

S -> i C t S | i C t S e S | a
C -> b

After Left factoring this grammar becomes

S -> i E t S S' | a
S' -> es | ϵ
E -> b

1. Recursive descent parser:

Recursive descent parsing is a top down parser method of syntax analysis in which we execute a set of recursive procedures to process the input. In recursive descent parsing, one procedure is associated with each non-terminal of the grammar.

Grammar must be non-left recursive and unambiguous.

Example:

Design a Recursive-Descent Parser for the grammar

E -> TE'

E' -> +TE' |ϵ
T -> FT'
T' -> * FT' |ϵ
F -> (E) | id

Solution:Pseudo Code of procedures for the above grammar are as follows:

```
procedure E( )
begin
     T();
     EPRIME();
end;
procedure EPRIME( )
begin
     if input_symbol ='+' then
     begin
        ADVANCE();
        T();
        EPRIME();
     end;
end;
procedure T( )
begin
     F();
     TPRIME();
end;
procedure TPRIME( )
begin
     if input-symbol ='*' then
     begin
           ADVANCE();
           F();
           TPRIME();
     end;
end;
procedure F( )
begin
     if input_symbol == 'id' then
             ADVANCE();
             else
```

```
                if input_symbol ==’(‘ then
                begin
                        ADVANCE();
                        E();
                        if input_symbol =’)’ then
                        ADVANCE()
                        else ERROR ();
                end;
                else
               ERROR ();
end;
```

2.Predictive parser

Predictive parser is a LL(1) type parser as it reads one token ahead of time from input before taking the decision for the production to be taken. In recursive descent parser stack is implicitly used while predictive parser uses on explicit stack. Model of predictive parser is shown below

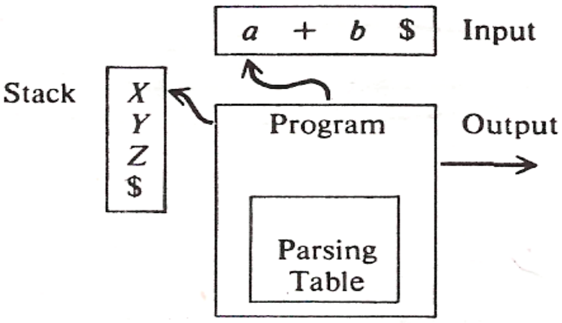


Figure: Model of a Predictive parser

The predictive parser has an input, a stack, a parsing table and an output. The input contains the string to be parsed followed by $ (the ***right end marker)***. The stack contains a sequence of grammar symbols, preceded by $ (the ***bottom-of-stack*** marker). Initially, the stack contains the start symbol of the grammar preceded by $.

The parsing table is a two-dimensional array M [A, a] where ‘A’ is non-terminal and ‘a’ is a terminal or the symbol $.

Working

The parser determines ‘X’ , the symbol on top of the stack and ‘a’ , the current input symbol. There are three cases possible:

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is non-terminal, the program consults entry M [X, a] of the parsing table M. This entry will have one of the production of X or an error entry. If M [X, a] = [X -> UVW] the parser replaces X on top of the stack by UVW (with U on top). It pushes W,followed by V and finally U.

If M [X, a] = error, the parser calls an error recovery routine.

**Refer Class Notes for example*

Construction of parsing table

To fill in the entries of a predictive parsing table, we need two functions along with grammar G. They are *FIRST and FOLLOW.*

Consider an example grammar for producing predictive parser table

E -> TE'
E' -> +TE' |ϵ
T -> FT'
T' -> * FT' |ϵ
F -> (E) | id

Set FIRST (X) for all non-terminals X of grammar G

FIRST (X) is the set of terminals that occur in the beginning of strings derived from X. To compute FIRST(X) apply the following rules

1. If X -> aα is a production then add 'a' to FIRST(X).
2. If X -> Yα is a production then add FIRST (Y) to FIRST(X).
   If Y -> ϵ is a production then also add FIRST(α) to FIRST(X). Assuming if α begins with Z then add FIRST(Z) to FIRST(X).

For the above example the FIRST set are as follows

FIRST(E) = FIRST(T) = FIRST(F) = { (, id}
FIRST(E') = {+}
FIRST(T') = {*}

Predictive parser table so far...

	id	+	*	(	)	$
E	E -> TE'			E -> TE'		

E’		E’-> +TE’				
T	T -> FT’			T -> FT’		
T’			T’-> * FT’			
F	F -> id			F -> (E)		

Set FOLLOW (X) for all non-terminals X of grammar G

FOLLOW (X) is the set of terminals that appear immediately to right of ‘X’ in some sentential from, i.e. terminals that can succeed a string that can be derived from X in some sentence of the language defined by the grammar. To compute FOLOW(X) apply the following rules

1. If A -> α Xaβ is a production then add 'a' to FOLLOW(X).
2. If A -> α XYβ is a production then add FIRST(Y) to FOLLOW(X).

If Y -> ϵ is a production then also add FIRST(β) to FOLLOW(X). Assuming if β begins with Z then add FIRST(Z) to FOLLOW(X).

3. If A -> αX ia a production then add FOLLOW(A) to FOLLOW(X)
4. If X is a start symbol (say S) then add $ to FOLLOW(S).

For the above example the FOLLOW set are as follows

FOLLOW(E) = FOLLOW(E’) = {), $}

FOLLOW(T) = FOLLOW(T’) = {+,), $}

FOLLOW(F) = {*,+,), $}

Final Predictive parser table

	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’-> +TE’			E’ -> ϵ	E’ -> ϵ
T	T -> FT’			T -> FT’		
T’		T’ -> ϵ	T’-> * FT’		T’ -> ϵ	T’ -> ϵ
F	F -> id			F -> (E)		

**Refer Class Notes for example string*

What should we do if parsing table has multiply defined entries?

- Eliminate left recursion

- Eliminate ambiguity
- Left factor the grammar

Unfortunately not all CFG are LL(1) grammar hence not all CFG can be recognized by recursive and predictive parser.

Bottom-up Parsing

This parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).

1. Shift Reduce Parsing technique (Shift Reduce Parser)

Handles:A handle is a right sentential form γ is a production A -> β in the position of γ where the string β may be found and replaced by A to produce the previous right sentential form in a rightmost derivation of γ. That is if S==> αAw, then A->β in the position following a is a handle of $\alpha\beta$w. The string w to right of the handle contains only terminal symbols.

1. Shift Reduce Parsing technique (Shift Reduce Parser)

his parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).It is like reducing a string "w" to start symbol of a grammar. Hence the derivation will now be reversed called reduction. Replacement of right hand side of the production by the left side is called ***reduction.***

A ***handle*** is a substring made up of terminals and non-terminals (say β) of a valid right sentential form (say γ) that can be replaced by its left sentential using a production (say A -> β that replaces β in γ by A) that eventually leads to one step of reduction towards the start symbol i.e. the previous right sentential form. This process is called ***handle pruning.***

Example: If S-> $\alpha A\delta$ and A -> β are productions then in reduction process of any string w that can be derived from S if the right sentential form is $\alpha\beta\delta$ it can be replaced by $\alpha A\delta$ which is its left sentential form. This handle will further reduce to S.

Shift-Reduce Parser is LR(0) parser i.e. it will scan the input string from left to right and perform reduction based on rightmost derivation without reading next symbol from input.

Example: Consider the grammar

S -> CC
C -> aC | b

For a string “aabab”

Rightmost Derivation	Canonical Reduction (Reverse of RMD)
S -> CC	aabab
=> CaC	aaCab
=> Cab	aCab
=> aCab	Cab
=> aaCab	CaC
=> aabab	CC
	S

Note from the above example that handle pruning in LR parsers are of type $\alpha\beta w$ *replaced by* αAw *where w is a string of only terminals.*

Working

A shift reduce parser uses a stack and an input buffer. $ is used to mark bottom of stack and right end of input as shown below

Stack	**Input**
$	**w $**

At every step the parser performs one of the two actions Shift or Reduce until an error is encountered or string is accepted.

Shift action: The next token/terminal/symbol from input buffer is PUSHed on top of stack.

Reduce action: The parser has detected a handle on top of stack which is to be replaced by appropriate production's left side to perform handle pruning/ reduction to previous right sentential form.

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to left side of appropriate production (say A -> β). It pops β from top of stack and replaces it by A on top of the stack. The parser repeats this process until it has detected an error or until the stack contains the start symbol and the input is empty as shown

Stack	**Input**
$ S	**$**

If the parser halts in the above condition it announces successful completion of parsing i.e. the string is accepted.

Two Problems –

- How to locate a handle in a right sentential form.
- *What production to choose in case there is more than one production with the same right side.*

2. Operator procedure parsing -

A grammar whose no production's right side is Є and does not have two adjacent non-terminals in called an operator grammar.

E.g. E • EAE | (E) | -E | id

A • + | – | * | / | •
Is not operator grammar

E • E+E | E–E | E * E | E/E | E • E | (E) | – E | id
is an operator grammar.

Operator precedence parsing uses three disjoint precedence relations :–
<·, =, and ·>

These precedence relations guide in selection of a handle.
a <· b • a " yields precedence to " b

a = b • a" has the same precedence as "b

a ·> b • a " takes precedence over " b
Traditional motions of associatively and precedence of operators is used to determine what precedence relation should hold between a pair of terminals.

*Refer Class Notes for examples.

PROCESS –

1. *Scan the string from the left end until the leftmost ·> is encountered.*
2. *Then scan backwards (to the left) over any ='s until a <· is encountered.*
3. *The handle contains everything to the left of the first ·> and to the right of the <· encountered, including any intervening or surrounding non-terminals.*

3. LR (k) parsers –

LR (k) parsing :

- *"L" is for left - to - right scanning of the input,*
- *"R" for constructing a rightmost derivation in reverse and 'k' for the input symbols of lookahead that are used in making parsing decisions.*
- *When 'k' is omitted, k is assumed to be 1.*

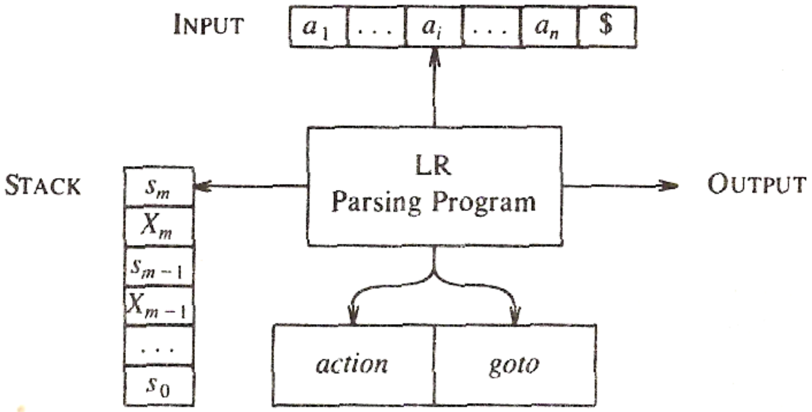


Fig. Model of an LR(k) parser

LR parser consists of an input; an output, a stack, a driver program and a parsing table that has two parts (action and goto)
The driver program is same for all LR parsers only parsing table changes from one parser to another.

There are three techniques for constructing LR parsing table
a) Simple LR b) Canonical LR c) Lookahead LR

The first method, simple LR (SLR) is easiest to implement but least powerful.

Second method, canonical LR is most power and most expensive.

The third method called lookahead LR (LALR) is intermediate in power and cost between the other two.

Construction of simple LR parsers [SLR i.e. LR (O)] -

To construct LR (0) collection for a grammar, we define augmented grammar and two functions closure and goto.
If G is a grammar with start symbol S, then G’, the augmented grammar for G, is G with a new start symbol S’ and production S’ • S.
The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of input.
i.e. acceptance occurs when and only when the parser is about to reduce by S’ • S.

Construction of LR (0) parser -

Step1 : Convert the grammar into augmented form –

Step 2 : Canonical collection of LR (O) items –

{RULES -

'•' followed by non-terminal write all rules of that non-terminal with '•' at the beginning.

'•' preceded by non-terminal, which has left recursive, production add it to the item.

NOTE: '•' acts as a separator, anything on L.H.S> of '•' are available on the stack and anythi on R.H.S is expected after that.

}

Step 3: Go to Graph

Step 4 : Construct parsing table

Explain with an example

* Refer Class notes

Construction of LR (1) parser -

* Refer Class notes

Construction of LALR parser –

Step 1: Convert the grammar into augmented form

Step 2: Construct LR (1) parser

Step 3: Try to merge the states for which there is no conflict.

Explain with an example

- Refer Class notes

Distinguish between top-down and bottom-up parsers

	Top-down parsers	Bottom-up parsers
(1)	Parser of this type generate the parse tree from the root to the leaves.	Parser of this type generate the parse tree from the leaves to the root.
(2)	When input is left to right, the parser is based on leftmost derivation. Hence called LL type.	When input is left to right, the parser is based on rightmost derivation. Hence called LR type.
(3)	It may cause backtracking.	It does not cause backtracking.

(4)	Left recursion is to be eliminated before designing top-down parser.	No problem of left recursion in bottom-up parser.
(5)	No handle detection problem.	Handle detection is a problem.
(6)	Example: Recursive-Descent Parser Predictive Parser	Example: Shift-Reduce Parser Operator-Precedence Parser.

<u>University questions</u>

1) What is Parsing? Differentiate top-down v/s bottom-up parsing method.

IT Dec04 (08M)

2) Distinguish between top-down and bottom-up parsing.

CMPN May04 (05M)

3) What is grammar? Explain different types of grammars with the help of example.

CMPN Dec04 (10M),IT Jun08 (10M)

4) Explain Recursive Descent Parser.

CMPN Dec05 (05M),May06 (06M),Dec07 (10M),Dec08 (10M)

5) Consider the following CFG

E -> E + T / T

T -> T * F/F

F -> (E) / I

I -> a / b / c

Remove the left recursion from the grammar

CMPN May07 (10M)

6) Consider the following CFG

S -> A

A -> Ad / Ae / aB / aC

B -> bBC / f

C -> g

Eliminate left recursion from the grammar

CMPN Dec08 (10M)

7) With example explain the process of elimination of Left Recursion.

CMPN May10 (05M)

8) Define Handle with suitable example.

CMPN Dec04 (02M)

9) Explain Shift Reduce parsing in detail.

IT May05 (06M)

10) Consider the following grammar:

E -> E + T

E -> T
E -> T * F
T -> F
T -> (E)
F -> id
Show the Shift Reduce parser action for the string id+id+id*id.

CMPN May07 (10M)

11) Explain operator precedence parser with suitable example.

CMPN May06 (10M),Dec06 (10M),Jun08 (10M)

12) With the help of the following grammar and given string explain the role of operator precedence parser.

E -> E + T / T
T -> T * V / V
V -> a /b/c/d
String to parse "a+b*c*d".

CMPN May10 (10M)

13) Explain LALR parsing.

CMPN May04 (05M)

Syntax directed translation

University questions

1. Write notes on Syntax directed translation.

CMPN May05 (05M)

2. Explain Syntax directed transition. Give Syntax directed definition to translate infix expression to postfix expression.

CMPN Dec05 (10M),May07 (10M),June 08 (10M)

3. Explain syntax directed translation with respect to construction of syntax tree.

CMPN May10 (10M)

Run time storage

Intermediate code generation

Intermediate Languages

Syntax trees and postfix notation are two kinds of intermediate representations. A third, called three-address code, will be used in this chapter. The semantic rules for generating three-address code from common programming language constructs are similar to those, for constructing syntax trees or for generating postfix notation.

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A dag gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement a : = b* – c + b* – c appear in Fig.

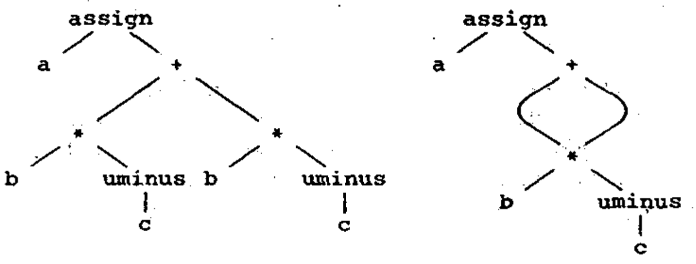


Fig.: Graphical representations of a : = b* – c + b* – c

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

a b c. uminus * b c uminus * + assign

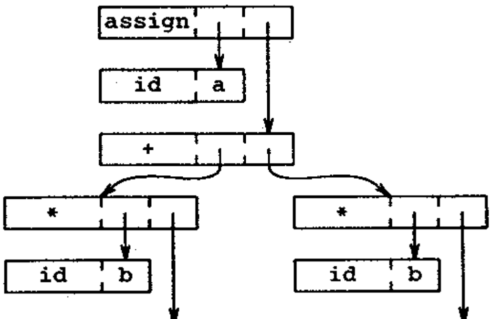
The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered from the order in which the nodes appear and the number of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a stack, of an expression in postfix notation. The relationship between postfix notation and code for a stack machine.

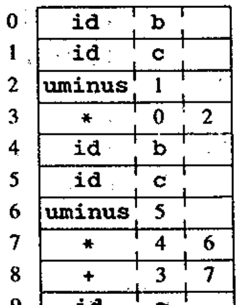
Syntax trees for assignment statements are produced by the syntax-directed definition in fig. Nonterminal *S* generates an assignment statement. The two binary operators + and *; are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree of Fig. from the input a : = b* – c + b* – c.

PRODUCTION	SEMANTIC RULE
S→ id : = E	S.nptr := mknode{'assign', mkleqf (**id**, **id.**p!aee), E.nptr)
E → E_1 + E_2	E.nptr := mknode{'+', Ey.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr := mknode('*', E_1.nptr, E_2.nptr)
E → – E_1	E.nptr := mkunode ('uminus^r, E_1.itptr)
E → – (E_1)	E.nptr := E_1.nptr ,
E → **id**	E.nptr := mkleaf (**id**, **id.**place)

Fig.: Syntax-directed definition to produce syntax trees for assignment statements.

This same syntax-directed definition will produce the dag , if the functions mknode(op, child) and mknode(op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes.
The token id has an attribute place that points to the symbol-table entry for the identifier.
Two representations of the syntax tree in fig. Each node is represented as a record with a field for its operator and additional fields for pointers to its children.





0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7

Fig. : Two representations of the syntax tree

Three-Address Code

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

x : = y op z

where x, y, and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Note-that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x + y * z might be translated into a sequence

t_1 : = y * z

t_2 : = x + t_1

where t_1 and t_2 are compile-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flbw-of-control statements makes three-address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged — unlike postfix notation.

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

t_1	:	=	–c	t_1	:	=	–c
t_2	:	=	b * t1	t_2	:	=	b * t_1
t_3	:	=	–c	t_5	:	=	t_2 + t_2
t_4	:	=	b * t3	a	:	=	t_5
t_5	:	=	t2 + t4				
a	:	=	t5				

(a) Code for the syntax tree. (b) Code for the dag.

Fig.: Three-address code corresponding to the tree and dag

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer to a symbol-table entry for that name.

Syntax-Directed Translation into Three-Address Code

Flow-of-control statements can be added to the language of assignments by productions and semantic rules like the ones for while statements. In the figure, the code for S → while E do S_1 is generated using new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code for *S*, respectively. These attributes represent labels created by a function *newlabel* that returns a new label every time it is called. Note that *S.afier* becomes the label of the statement that comes after the code for the while statement. We assume that a non-zero expression represents true; that is, when the value of *E* becomes zero, control leaves the while statement.

PRODUCTION	**SEMANTIC RULES**	
S→ id := E	S.code	: = E.code || gen(id.place ':=' E.place)
E → E_1 + E_2	E.place E.code	: = newtemp : = E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place E.code	: = newtemp : = E_1.code || E_2.code || gen (E.place ':=' E_1.place.'*' E_2.place)
E → – E_1	E.place E.code,	: =. newtemp; : = E_1.code || gen(E.place ':'=' 'uminus’ E_1.place
E → – (E_1)	E.place E.code	: = E_1.place; := E_1.code
E → id	E.place E.code	: = id.place; : = id.code

Fig. 8.6 : Syntax-directed definition to produce three-address code for assignments.

PRODUCTION	SEMANTIC RULS
S → while E do S1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ':') || E.code || gen ('if’ E.place ' = ' '0' 'goto' S.after) || S1.code || gen('goto'S.begin) || gen{S.after ':')

Fig. : Semantic rules generating code for a while statement.

Expressions that govern the flow of control may in general be boolean expressions containing relational and logical operators..

Implementations of three address statements :

A three address statement is an abstract form of intermediate code.
Three such representations are quadruples, triples and indirect triples.

1) Quadraples:

A quadraple is a record structure with four fields which we will call op, arg1, arg2 and result.
The contents of fields arg 1, arg 2 and result are normally pointers to the symbol table entries for the names represented by these fields.
Hence temporary names are to be entered into symbol table as they are created.

e.g. a : = b * - c + b * - c

producing intermediate code we get, optimized code

t_1 : = -c
t_2 : = b * t_1
t_3 : = - c
t_4 : = b * t_3
t_5 : = t_2 + t_4
a : = t_5

	Op	Arg 1	Arg 2	Result
(0)	Uminus	C		t_1
(1)	*	B	t_1	t_2
(2)	Uminus	C		t_3
(3)	*	B	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_5		a

2)Triples :

To avoid temporary names into the symbol table we might refer to a temporary value by the position of the statement that computes it.
Three address statement can be represented by records with only three fields : op, arg1 and arg2.

Op	Arg 1	Arg 2

(0)	Uminus	C	
(1)	*	B	(0)
(2)	Uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	Assign	A	(4)

A ternary operation like x [i] : = y or x : = y [i] requires two entries in the triple structure.

	Op	Arg 1	Arg 2
(0)	[ ] =	x	I
(1)	Assign	(0)	Y

	Op	Org 1	Org 2
(0)	= [ ]	Y	I
(1)	Assign	X	(0)

3)Indirect triples :

In this method we list pointers to triples rather than listing the triples themselves

e.g. a : = b * - c + b * - c

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

	Op	Arg 1	Arg 2
(14)	Uminus	C	
(15)	*	B	(0)
(16)	Uminus	C	
(17)	*	B	(2)
(18)	+	(1)	(3)
(19)	Assign	A	(4)

During optimizations statements are after moved around.

In triples notation moving a statement that defines a temporary value requires us to change all references to that statement in Arg 1 and Arg 2 arrays.

In indirect triples, statement can be moved by re-ordering the statement list. Pointers to temporary values refer to op-Arg1–Arg2 arrays which are not changed. Hence only these pointers need to be changed.

University questions

1. Describe the various forms of intermediate code used by compilers.

CMPN May05 (10M),Dec06 (10M),IT May04 (10M),Dec07 (10M)

2. Distinguish between Parse tree and Syntax tree. Explain Syntax Directed Translation.

CMPN May04 (10M)

3. Give the various forms of intermediate code used by the compiler.

CMPN Dec07 (10M),IT Dec 05 (10M)

4. What are the different intermediate code forms used in compiler? Explain.

   CMPN May06 (08M)

5. Differentiate between Parse tree and Syntax tree. Also explain what ambiguous grammar is Assume suitable grammar.

IT May05 (10M), June07 (20M), Dec07 (10M)

6. Explain Parse tree and Syntax tree with the help of suitable examples.

CMPN Dec04 (04M)

7. Differentiate between Parse tree and Syntax tree.

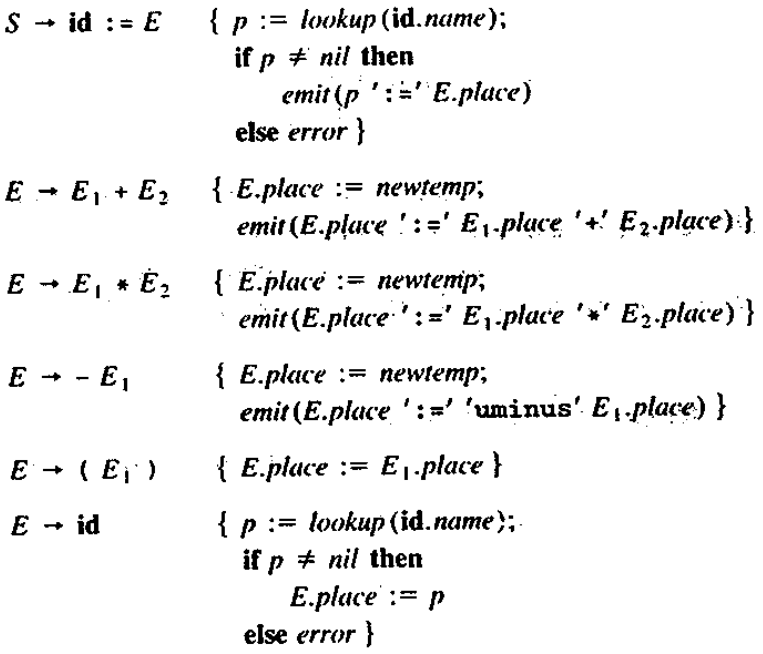
CMPN May10 (05M)

8. Generate three address code for a given expression

while (A < B) do
if (C < D) then X=Y+Z

Code generation

ASSIGNMENT STATEMENTS



```
S → id := E        { p := lookup(id.name);
                     if p ≠ nil then
                          emit(p ':=' E.place)
                     else error }

E → E1 + E2        { E.place := newtemp;
                     emit(E.place ':=' E1.place '+' E2.place) }

E → E1 * E2        { E.place := newtemp;
                     emit(E.place ':=' E1.place '*' E2.place) }

E → - E1           { E.place := newtemp;
                     emit(E.place ':=' 'uminus' E1.place) }

E → ( E1 )         { E.place := E1.place }

E → id             { p := lookup(id.name);
                     if p ≠ nil then
                          E.place := p
                     else error }
```

Fig. Translation scheme to produce three-address code for assignments.

BOOLEAN EXPRESSIONS

In programming languages, boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions. In turn, relational expressions, are of the form E_1 relop E_2, where E_1 and E_2 are arithmetic expressions.

We consider boolean expressions generated by the following grammar:

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$$

We use the attribute op to determine; which of. the comparison operators $<$, $\leq$, $=$, $\neq$, $>$, or $\geq$ is represented by relop. As is customary, we assume that or has lowest precedence, then and, then not.

Methods of Translating Boolean Expressions

There are two principal methods of representing the value of a boolean expression. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often 1 is used to denote true and 0 to denote false, although many other encodings are also possible. For example, we could let any nonzero quantity denote true and zero denote false, or we could let any nonnegative quantity denote true and any negative number denote false.

The second principal method of implementing boolean expressions is by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements. For example, given the expression E_1 or E_2, if we determine that E_1 is true, then we can conclude that the entire expression is true without having to evaluate E_2.

The semantics of the programming language determines whether all parts of a boolean expression must be evaluated. If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as E_1 or E_2 neither E_1 nor E_2 is necessarily evaluated fully. If either E_1 or E_2 is an expression with side effects (e.g., contains a function that changes a global variable), then an unexpected answer may be obtained.

Numerical Representation

Let us first consider the implementation of boolean expressions using 1 to denote true and 0 to denote false. Expressions will be evaluated completely, from left to right, in a manner similar to arithmetic expressions. For example, the translation for

a or b and not c

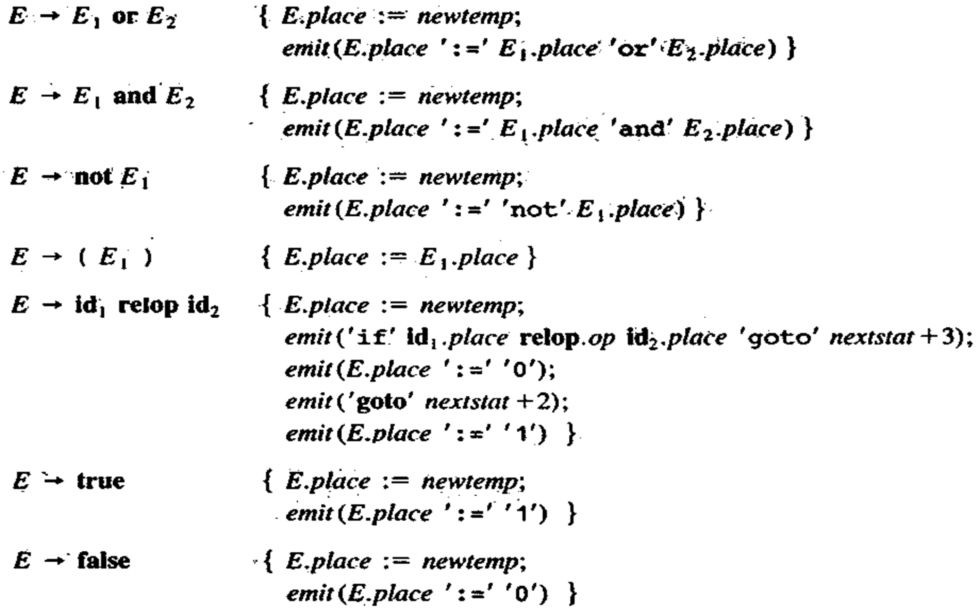
is the three-address sequence

t_1 : = not c
t_2 : = b and t1
t_3 : = a or t_2.

A relational expression such as a < b is equivalent to the conditional statement if a < b then 1 else 0, which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100):

```
100:   if a < b goto 103
101:   t : = 0
102:   goto 104
103:   t : = 1
104:
```

A translation scheme for producing three-address code for boolean expressions is shown in fig. In this scheme, we assume that emit places three-address statements into an output file in the right format, that nextstat gives the index of the next three-address statement in the output sequence, and that emit increments nextstat after producing each three-address statement.



```
E → E1 or E2            { E.place := newtemp;
                          emit(E.place ':=' E1.place 'or' E2.place) }

E → E1 and E2           { E.place := newtemp;
                          emit(E.place ':=' E1.place 'and' E2.place) }

E → not E1              { E.place := newtemp;
                          emit(E.place ':=' 'not' E1.place) }

E → ( E1 )              { E.place := E1.place }

E → id1 relop id2       { E.place := newtemp;
                          emit('if' id1.place relop.op id2.place 'goto' nextstat+3);
                          emit(E.place ':=' '0');
                          emit('goto' nextstat +2);
                          emit(E.place ':=' '1') }

E → true                { E.place := newtemp;
                          emit(E.place ':=' '1') }

E → false               { E.place := newtemp;
                          emit(E.place ':=' '0') }
```

Fig.: Translation scheme using a numerical representation for booleans.

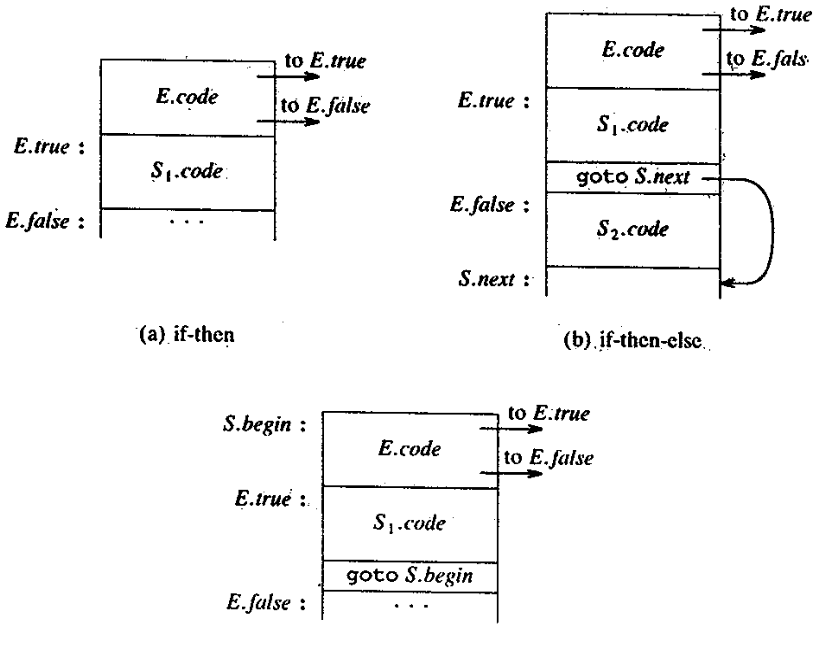
Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → if E then S
| if E then S_1 else S $_2$
| while E do S_1

In each of these productions, E is the boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.

With a boolean expression E, we associate two labels:
E.true the label to which control flows if E is true, and E.false, the label to which control flows if £is false. The semantic rules for translating a flow-of control statement-S allow control to flow from the translation S. code to the three address instruction immediately following S.code. In some cases, the instruction immediately following S.code is a jump to some label L. A jump to a jump to L from within S.code is avoided using inherited attribute S.next. The value of S.next is a label that is attached to the first three-address instruction of "> be executed after the code for S.
In translating the if-then statement S → if E then, a new label E.true is created and attached to the first three-address instruction generated for the statement S_1, as in Fig. The code for E generates a jump to E.true if E is true and a jump to S.next if E is false. We therefore set E.false to S.next.



(a) if-then

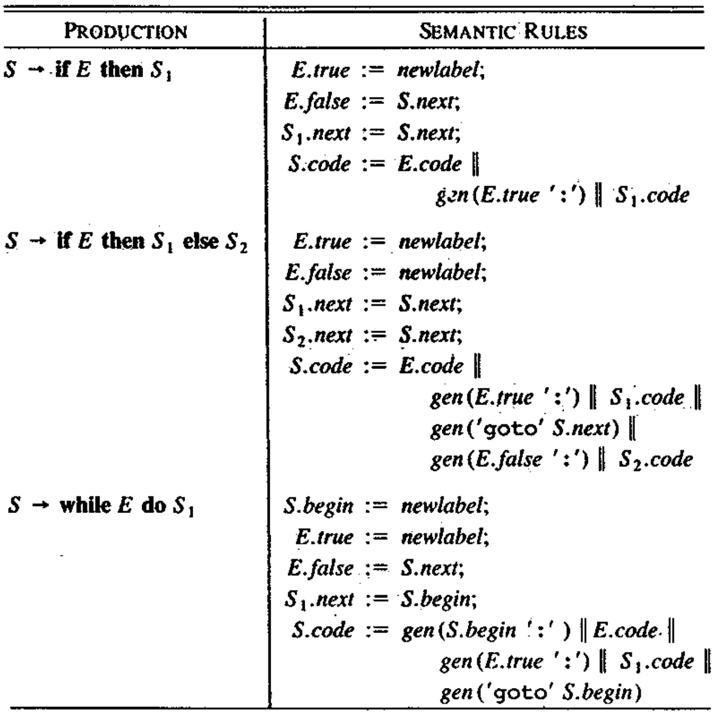
(b) if-then-else

(c) while-do

Fig. 8.22 : Code for if-then, if-thcn-elsc, and while-do statements.

In translating the if-then-else statement the code for the S →if E then S_1 else S $_2$; boolean expression E has jumps out of it to the first instruction of the code for S_1 if E is true, and to the first instruction of the code for S_2 if E is false. As with the if-then statement, an inherited attribute S.next gives the label of the three-address instruction to be executed next after executing the code for S. An explicit goto S.next appears after the code for S_l but not after S_2. We leave it to the reader to show that, with these semantic rules, if S.next is not the label of the instruction immediately following S_2.code then an enclosing statement will supply the jump to label S.next after the code for S_2.

The code for S → while E do S_1 while E do S_1 is formed as shown in Fig. A new label S.begin is created and attached to the first instruction generated for E. Another new label E.true is attached to, the first instruction S_1. The code for E generates a jump to this label if E is true, a jump to S.next if E is false; again, we set E.false to be S.next. After the code for S_1 we place the instruction goto S.begin, which causes a jump back to the beginning of the code for the boolean expression. Note that S.next is set to this label S.begin; so jumps from within S_1.code can go directly to S.begin.



PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true$:= $newlabel$; $E.false$:= $S.next$; $S_1.next$:= $S.next$; $S.code$:= $E.code$ ∥ $gen(E.true$ ':') ∥ $S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true$:= $newlabel$; $E.false$:= $newlabel$; $S_1.next$:= $S.next$; $S_2.next$:= $S.next$; $S.code$:= $E.code$ ∥ $gen(E.true$ ':') ∥ $S_1.code$ ∥ gen('goto' $S.next$) ∥ $gen(E.false$ ':') ∥ $S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin$:= $newlabel$; $E.true$:= $newlabel$; $E.false$:= $S.next$; $S_1.next$:= $S.begin$; $S.code$:= $gen(S.begin$ ':') ∥ $E.code$ ∥ $gen(E.true$ ':') ∥ $S_1.code$ ∥ gen('goto' $S.begin$)

Fig. : Syntax–directed definition for flow–of–control statements.

We discuss the translation of flow-of-control statements in more detail in Section 8.6 where an alternative method, called "backpatching," emits code for such statements in one pass.

Example:
Consider the statement

```
while a < b do
        if c < d then
                x := y + z
        else
                x : = y - z
```

The syntax-directed definition above, coupled with schemes for assignment statements and boolean expressions, would produce the following code:

```
L1: if a < b goto L2
        goto Lnext
L2: if c < d goto L3
        goto L4
L3:  t1  : = y + z
         x := t,
         goto L1
L4:  t2 := y - z
        x := t2
        goto L1
Lnext:
```

We note that the first two gotos can be eliminated by changing the directions of the tests.

BACKPATCHING

The easiest way to implement the syntax-directed definitions in above examples is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations given in the definition. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. We can get around this problem by generating a series of branching statements with the targets of the jumps temporarily left unspecified. Each such statement- will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

Let us see how backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass.
To manipulate lists of labels, we use three functions:

1. makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
2. merge(p_1, p_2) concatenates the lists pointed to by p_1 and p_2 and returns a pointer to the concatenated list.
3. backpatch(p, i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions

We now construct a translation' scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. We insert a marker nontermi^ nal *M* into the grammar to cause a semantic action to pick up, at appropriate times, the index of the next quadruple to be generated. The grammar we use is the following:

(1) E → E_1 or M E_2
(2) | E_1 and M E_2
(3) | not E_1
(4) | (E_1)
(5) | id_1 relop id_2
(6) | true
(7) | false
(8) M → ε

Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping code for boolean expressions. As code is generated. for E, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by E.truelist and E.falselist, as appropriate.

The semantic actions reflect the considerations mentioned above. Consider the production E → E_1 and M E_2. If E_1 is false, then E is also false, so the statements on E_1.falselist become part of E.falselist. If E_1 is true, however, we must next test E_2, so the target for the statements E_1.truelist must be the beginning of the code generated for E_2. This target is obtained using the marker nonterminal M. Attribute M.quad records the number of the first statement of E_2.code. With the production M → ε we associate the semantic action

{M. quad : = nextquad}

The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E_x.truelist when we have seen the remainder of the production E → E_1 and M E_2. The translation scheme is as follows.

(1) E → E_1 or M E_2 {backpatch (E_1. falselist, M.quad);
E.truelist : = merge (E_1.truelist, E_2.truelist);
E.falselist := E_2.falselist}

(2) E → E_1 and M E_2 {backpatch (E_1.truelist; M.quad);
E.truelist := E_2.truelist;
E.falselist := merge (E_1.falselist, E_2.falselist)}

(3) E → not E_1 {E.truelist := E1, falselist;
E.fatselist := E1.truelist

(4) E → (E_1) {E.truelist := E_1.truelist,
E.falselist := E_1.falselist}

(5) E → id_1 relop id_2 {E.truelist := makelist (nextquad);
E.falselist : = makelist (nextquad + 1);
emit{'if' id_1.place relop.op id_2.place 'goto_')
emit ('goto __')}

(6) E → true {E.truelist := makelist (nextquad);
emit ('goto __')}

(7) E → false {E.falselist := makelist (nextquad);
emit ('goto __')}

(8) M → ε {M.quad : = nextquad}

For simplicity, semantic action (5) generates two statements, a conditional goto and an unconditional one. Neither has its target filled in. The index of the first generated statement is made into a list, and E.truelist is given a pointer to that list. The second generated statement goto __ is made into a list and given to E.falselist.

Example:

Consider again the expression a < b or c < d and e < f. An annotated parse tree is shown in Fig. The actions are performed during a depth-first traversal of the tree. Since all actions appear at the ends of right sides, they can be performed in conjunction with reductions during a bottom-up parse., In response to the reduction of a < b to E by production (5), the two quadruples ;

100: if a < b goto _
101: goto _

are generated. The marker nonterminal M in the production E → E_1 or M E_2 records the value of nextquad, which at this time is 102. The reduction of c < d to E by production (5) generates the quadruples

102: if c < d goto _
103: goto _

We have now seen E in the production E → E_1 and M E_2. The marker nonterminal in this production records the current value of nextquad, which is now 104. Reducing e < f into E by production (5) generates

104: if e < f goto _
105: goto _

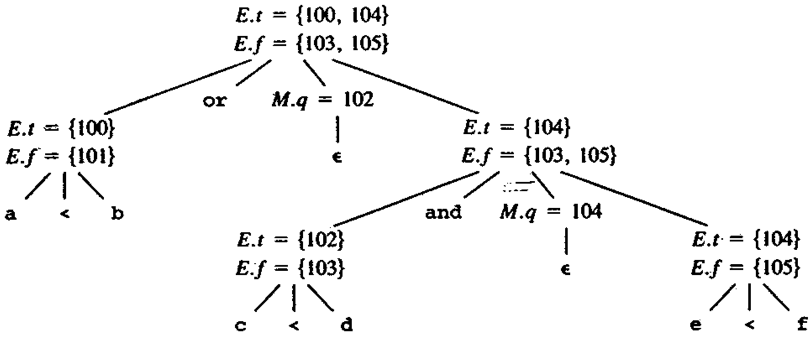


Fig. : Annotated parse tree for a<b or c<d and e<£.

We now reduce by E → E_1 and M E_2. The corresponding semantic action calls backpatch({102},104) where {102} as argument denotes a pointer to the list containing only 102, that list being the one pointed to by (E).truelist. This call to backpatch fills in 104 in statement 102. The six statements generated so far are thus:

100 : if a < b goto _
101 : goto _
102 : if c < d goto 104

```
        103 :   goto _
        104 :   if e < f goto _
        105 :   goto _
```

The semantic action associated with the final reduction by $E \rightarrow E_1$ or $M\ E_2$ calls backpatch({l01}, 102) which leaves the statements looking like:

```
        100 :   if a < b goto _
        101 :   goto 102
        102 :   if c < d goto 104
        103 :   goto _
        104 :   if e < f goto _
        105 :   goto _
```

The entire expression is true if and only if the goto's of statements 100 or 104 are reached, and is false if and only if the goto's of statements 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression.

Flow-of-Control Statements

We now show how backpatching can be used to translate flow-of-control statements in one pass. As above, we fix our attention on the generation of quadruples, and the notation regarding translation field names and list-handling procedures from that section carries over to this section as well. As a larger example, we develop a translation scheme for statements generated by the following grammar:

```
(1) S → if E then S
(2)             |       if E then S else S
(3)             |       while E do S
(4)             |       begin L end
(5)             |       A
(6) L →          L ;    S
(7)             |       S
```

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. Note that there must be other productions, such as those for assignment statements. The productions given, however, will be sufficient to illustrate the techniques used to translate flow-of-control statements.

Our general approach will be to fill in the jumps out of statements when their targets are found. Not only do boolean expressions need two lists of jumps that occur when the expression is true and when it is false, but statements also need lists of jumps (given by attribute nextlist) to the code that follows them in the execution sequence.

Scheme to Implement the Translation

The nonterminal E has two attributes E.truelist and E.falselist, as above. L and S each also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes L.nextlist and S.nextlist. S.nextlist is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and L.nextlist is defined similarly.

In the code layout for S → while E do S_1 in Fig., there are labels S.begin and E.true marking the beginning of the Code for the complete statement S and. the body S|. The two occurrences of the marker nonterminal M in the following production record the quadruple numbers of these positions:

$$S \rightarrow \text{while } M_1 \text{ E do } M_2 \text{ } S_1$$

Again, the only production for M is M → ε with an action setting attribute M.quad to the number of the next quadruple. After the body S_1 of the while statement is executed, control flows to the beginning. Therefore, when we reduce while M_l E do M_2 S_1 to S, we backpatch S_1.nextlist to make all targets on that list be M_1.quad. An explicit jump to the beginning of the code for E is appended after the code for S because control may also "fall out the bottom." E.truelist is backpatched to go to the beginning of S_1 by making jumps on E.truelist go to M_2.quad.

A more compelling argument for using S.nextlist and L.nexttist comes when code is generated for the conditional statement if then S_1 else S_2. If control "falls out the bottom" of S_1, as when S_1 is an assignment, we must include at the end of the code for S_1 a jump over the code for S_2. We use another marker nonterminal to introduce this jump after S_1. Let nonterminal N be this marker with production N → ε. N has attribute N.nextlist, which will be a list consisting of the quadruple number of the statement goto _ that is generated by the semantic rule for N. We now give the semantic rules for the revised grammar.

(1) S → if E then M_1 S_1 N else M_2 S_2
{backpatch (E.truelist, M_1 .quad);
backpatch (E.falselist, M_2.quad);
S.nextlist := merge (S_1.nextlist, merge(N.nextlist, S_2.nextlist))}

We backpatch the jumps when E is true to the quadruple M .quad, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S_2. The list S.nextlist includes all jumps out of S_1 and S_2, as well as the jump generated by N.

(2) N → ε {N.nextlist := makelist (nextquad);
emit ('goto _')

(3) M → ε {M.quad : = nextquad }

(4) S → if E then M S_1 {backpatch {E.truelist, M.quad);
S.nextlist := merge
(E.falselist,S.nextlist)}

(5) S → while M_1 E do M_2 S_1 {backpatch (S.nextlist, M_1.quad);
backpatch (E.truelist, M_2.quad);

S.nextlist = E.falselist
emit ('goto' M. quad)}

(6) S → begin L end {S.nextlist = L.nextlist}

(7) S → A {S.nextlist := nil}

The assignment S.nextlist := nil initializes S.nextlist to an empty list.
(8) L → L_1 ; M S {backpatch (L_1.nextlist, M.quad);
L.nextlist : = S.nextlist}

The statement following L\ in ofder of execution is the beginnmg of S. Thus the L.nextlist list is backpatched to the beginning of the code for S, which is given by M.quad.

(9) L → S { L.nextlist=S.nextlist }

Note that no new quadruples are generated anywhere in these semantic rules except for rules (2) and (5). All other code is generated by the semantic actions associated with assignment statements and expressions. What the flow of control does is cause the proper backpatching so that the assignments and boolean expression evaluations will connect properly.

PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls, and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement.
(1) S → call id (Elist)
(2) Elist → Elist , E
(3) Elist → E

Calling Sequences

When a procedure call occurs, space must be allocated for the activation record of the called procedure. The argument's of the called procedure must be evaluated and made available to the called procedure in a known place. Environment pointers must be established to enable the called procedure to access data in enclosing blocks. The state of the calling procedure must be saved so it can resume execution after the call. Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished. The return address is usually the location of the instruction that follows the call in the calling procedure. Finally, a jump to the beginning of the code for the called procedure must be generated.

When a procedure returns, several actions also must take place. If the called procedure is a function, the result must be stored in a known place. The activation record of the calling procedure must be restored. A Jump to the calling procedure's return address must be generated.

There is no exact division of the un-time tasks between the calling and called procedure. Often the source language, the target machine, and the operating system impose requirements that favor one solution over another.

A Simple Example

Let us consider a simple example in which parameters are passed by reference and storage is statically allocated. In this situation, we can use the param statements themselves as placeholders for the arguments. The called procedure is passed a pointer in a register to the first of the param statements, and can obtain pointers to any of its arguments by using the proper offset from this base pointer. When generating three-address code for this type of call, it is sufficient to generate the three-address statements needed to evaluate those arguments that are expressions other than simple names, then follow them by a list of param three-address statements, one for each argument. If we do not want to mix the argument-evaluating statements with the param statements, we shall have to save the value of E.place, for each expression E in id (E, E, ……., E).

A convenient data structure in which to save these values is a queue, a first-in first-out list. Our semantic routine for Elist → Elist, E will include a step to store E.place on a queue. Then, the semantic routine for

S → call id (Elist) will generate a param statement for each item on queue, causing these statements to follow the statements evaluating. the argument expressions. Those statements were generated when the arguments themselves were reduced to E The following syntax-directed translation incorporates these ideas.

```
(1)   S → call id  (Elist)
                                    {   for. each item p on queue do
                                                  emit('param' p);
                                        emit('call' id.place)}
```

The code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement. A count of the number of parameters is not generated with the call statement but could be calculated in the same way we computed Elist.ndim in the previous section.

```
(2) Elist → Elist , E
                                    {append E.place to the end of queue}
(3) Elist → E
                                    {initialize queue to contain only E.place}
```

Here queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

Code Optimization

➢ Code optimization:

Machine independent optimization:

1.Folding:

Folding is replacement of expressions that can be evaluated at compile by their computed values.

i.e. If the values of all the operands of an expression are known at compile time, the expression can be replaced by its computed value.

```
e.g.   A = 2 * 3 + A + C
       could be replaced by
       A = 6 + A +C
```

2. Constant propagation:

Constant propagation is folding applied in such a way that values know at compile time to be associated with variables are used to replace certain uses of these variables in the translated program text.

```
e.g.   The program fragment is –
       PI = 3.141592
       D_To_R = PI/ 180.0
       Can be written as
       D_To_R = 3.141592 / 180.0
       Further more another application of folding eventually yields.
       PI = 3.141592
       D_To_R = 0.0174644
```

3.Redundant subexpression elimination-

An expression once computed is reused rather than recomputed.

This will reduce the no. of operations that must be performed when the program is eventually executed.

```
e.g.   x = C + 3 + A + 5
       y = 2 + A + 4 + C
       can be rearranged to give
       x = 3 + 5 + A  + C
       y = 2 + 4 + A + C
       which gives
       T = A + C
       X  = 8 + T
       Y = 6 + T
```

4.Loop unrolling –

It is an optimization technique in which the code constituting the loop body is reproduced a number of times rather than just once.

e.g. Simple initialization loop

```
        For (i = 0; i < 30; i++)
        A (j + i) = 0.0;
```

Loop can be transformed into

```
        A (j + 0) = 0.0;
        A (j + 1) = 0.0;
        A (j + 2) = 0.0;
                .
                .
                .
        A (j + 29) = 0.0;
```

5.Frequency reduction:

It is an optimization technique that moves certain computations from program regions where they are very frequently executed to the regions where they are less frequency executed.

Especially frequency reduction is implemented with loop structures by moving certain invariant operations out of the loop body and placing them in a block just prior to the start of the loop.

```
e.g.    While (i < = limit – 2)
        {       .
                .
        }
```

evaluation of limit –2 is a loop invariant computations in the above while statement.

The equivalent code is,

```
        t = limit –2;
        while (i < = t)
        {       .
                .
        }
```

6.Strength reduction:

It is an optimization technique which consist of the replacement of one type of operation by another operation that takes less time to execute.
e.g. An e.g. of strength reduction is replacement of multiplication by addition.

Machine dependent optimization :

When code is modified to make it more efficient on a particular machine, machine – dependent optimization is being performed.
Machine dependent optimization uses information about the limits and special features of the target machine to produce code which is shorter or which executes more quickly on the machine.

e.g. Consider a code intended for machines of PDP-II family.
1) These computers have auto increment and auto decrement modes of instructions.
When an instruction is given in auto increment mode, the contents of the register are incremented after being used.
The register is incremented by one for byte instruction and by two for word instructions.
The uses of instructions in these modes reduces the code necessary for pushing and popping stacks.

2) PDP-11 computer have machine level instructions to increment (INC) or to decrement (DEC) by one, values stored in memory.
Whenever possible, the INC and DEC operations should be used instead of creating a constant with value 1 and adding or subtracting this constant form the value stored in memory.

3) PDP-11 machines have left and right shift operations shifting the bits one position to the left is equivalent to multiplying by 2.

Since shifting is faster than multiplication or division, hence more efficient code is generated if shifting is used.

4) If two instructions do not depend on one another and if the target machine allows parallel processing, the compiler should generate code to take advantage of this capability.

5) When different instructions are available to perform a task, the most efficient should be chosen.

THE PRINCIPAL SOURCES OF OPTIMIZATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving transformations.
Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B_5 shown in Fig. recalculates $4*i and 4*j$.

B_5

t_6
: =
x
: = a [t_6]
t_7
: =
t_8

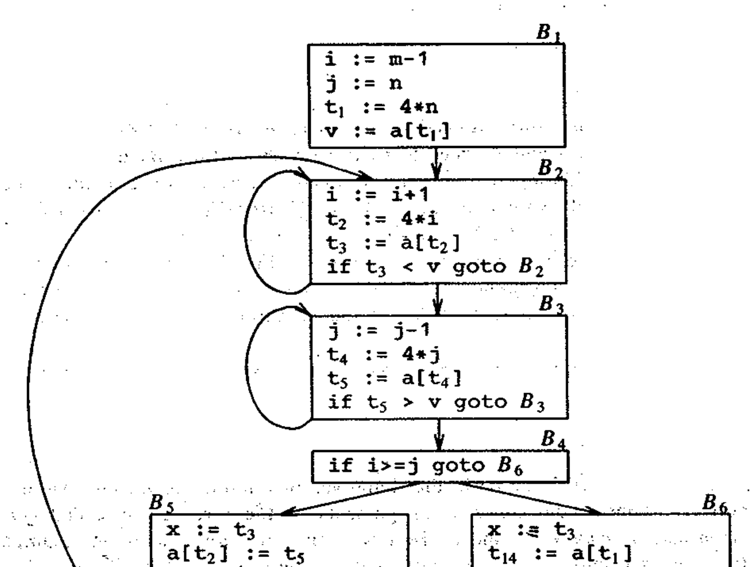
B_5

t_6
: =
x
: = a [t_6]
t_8
: =
t_9

Common Subexpressions

An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. For example, the assignments to t_7 and t_{10} have the common subexpressions $4*i$ and $4*j$, respectively, on the right side in Fig.
This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: Figure shows the result of eliminating both global and local common subexpressions from blocks B_5 and B_6 in the flow graph of Fig. We first discuss the transformation of B_5 and then mention some subtleties involving arrays.



After local common subexpressions are eliminated, B_5 still evaluates $4*i$ and $4*j$, as shown in Fig.. Both are common subexpressions; in particular, the three statements

$t_8 := 4*j$; $t_9 := a[t_8]$; $a[t_8] := x$

in B_5 can be replaced by

$t_9 := a[t_4]$; $a[t_4] := x$

using t_4 computed in block B_3: In Fig. 10.7, observe that as control passed : from the evaluation of $4*j$ in B_3 to B_5, there is no change in j, so t_4 can be used if $4*j$ is needed.

Another common subexpression comes to light in B_5 after t_4 replaces t_8. The new expression a[t_4] corresponds to the value of a [j] at the source level. Not only does j retain its value as control leaves B_3 and then enters B_5 but a [j], a value computed into a temporary t_5, does too because there are no assignments to elements of the array a in the interim. The statements

$t_9 := a[t_4]$; $a[t_6] := t_9$

in B_5 can therefore be replaced by

$a[t_6] := t_5$

Analogously, the value assigned to x in block B_5 of Fig. is seen to be the same as the value assigned to t_3 in block B_2. Block B_5 in Fig. is the result of eliminating common subexpressions corresponding to the values of the source level expressions a [i] and a [j] from B_5 in Fig. A similar series of transformations has been done to B_6 .

The expression a [t_1] in blocks B_1 and B_6 of Fig. is not considered a common subexpression, although t_1 can be used in both places. After control leaves B and before it reaches B_6, it can go through B_5, where there are assignments to a. Hence, a [t_1] may not have the same value on reaching B_6 as it did on leaving B_x, and it is not safe to treat a [t_1] as a common subexpression.

Copy Propagation

Block B_5 in Fig. can be further improved by eliminating x using two new transformations. One concerns assignments of the form f : = g called copy statements, or copies for short. For example, when the common subexpression in c : = d + e is eliminated in Fig. 10.8, the algorithm uses a new variable t to hold the value of d + e. Since control may reach c : = d + e either after the assignment to a or after the assignment to b, it would be incorrect to replace c : = d + e by either c: = a or by c: = b.

The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement f : = g. For example, the assignment x : = t_3 in block B_5 of Fig.10.7 is a copy. Copy propagation applied to B_s yields:

x : = t_3
a[t_2] : t_5
a[t_4] : = t_3
goto B_2

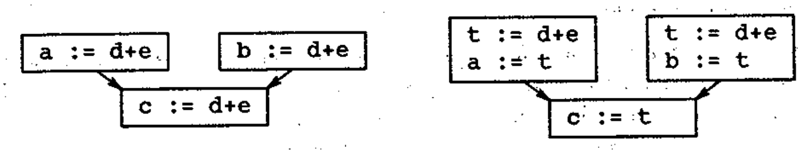


Fig. 10.8. Copies introduced during common subexpression elimination.

This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x.

Dead-Code Elimination

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms into:

a[t_2] : = t_5
a[t_4] : = t_3
goto B_2

Loop Optimizations

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: code motion, which moves code outside a loop; induction-variable elimination, which we apply to eliminate i and j from the inner loops B_2 and B_3 of Fig. and reduction in strength, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop.

For example, evaluation of limit–2 is a loop-invariant computation in the following while-statement:

```
while ( i ⇐ limit –2 ) / * statement does not change limit */
```

Code motion will result in the equivalent of t = limit-

```
while ( i ⇐ t ) /* statement does not change limit or t */
```

Compiler-Compilers

Parser generator YACC –

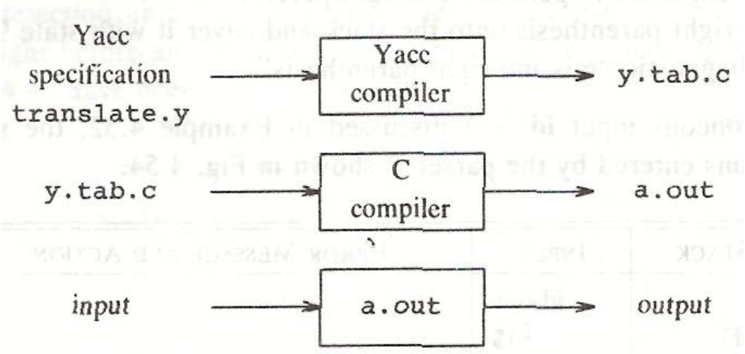


Fig. Creating an input/output translator with YACC.

YACC stands for "Yet another Compiler – Compiler ".
YACC is available as a command on the UNIX System and has been used to help implement hundreds of compilers.
A translator can be constructed using YACC in the manner shown in figure above.
A file say translate .y containing yacc specifications of the translator is prepared. The UNIX system command

```
yacc translate.y
```

transforms file translate.y into a c program y.tab.c using the LALR method.
The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that user may have prepared.

By compiling y.tab.c along with the 'ly' library that contains the LR parsing program using the command.

cc y.tab.c -ly

We obtain the desired object program a.out that performs the translation specified by the original YACC program.

A YACC source program has three parts –

declarations

% %

translation rules

% %

Declarations part

There are 2 optional structures in declaration part.

- ***Original C declarations delimited by % { and % }***
- ***% token DIGIT declares DIGIT to be token.***

Tokens declared in this section can then be used in second and third parts of YACC specification.

Translation rules par

In the part of YACC specification after the first %{ %} pair, we put translation rules.

Each rule consist of production and associated semantic action.

A set productions of the form

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

would be written in the YACC as

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

In a semantic action, the symbol $$ refers to the attribute value associated with the non-terminal on the left, while $I refers to the value associated with the ith grammar symbol (terminal or non-terminal) on the right.

The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for $ $ in terms of the $ i's e.g.

```
    % {
    # include < type.h>
    %}
    % token DIGIT

    % %
    line    :       expr  '\n'  { printf ( "%d \n" , $ 1 ) ; }
    ;

    expr   :        expr  '+' term { $ $ = S 1 + $ 3 ; }

                    | term
    ;
    term  :         term '*'  factor { $ $ = $ 1  *  $ 3 ; }

                    |factor
    ;
    factor          :       '(' expr  ')'   { $ $ = $ 2 ; }

                            | DIGIT
    ;
    % %

    yylex  ( )
   {
    }
```

Creating Yacc Lexical analyzers with Lex

LEX library provides driver program yylex ( ) i.e. name of lexical analyzer required by YACC.

Under UNIX system, if the LEX specification is in the file first.l and the YACC specification in second.y we can say-

```
Lex first.l
Yacc second.y
cc   y.tab.c  -ly  - l1
```

To obtain desired translator.

➢ ***JAVA COMPILER AND ENVIRONMENT:***

JAVA is a new programming language and operating environment developed by Sun Microsystems.

Java is an object-oriented language. The object orientation is Java is stronger than many other languages, such as C++. Except for a few primitive data types, everything in Java is an object. Arrays and strings are treated as objects. Even the primitive data types can be encapsulated inside objects. There are no procedure or functions in Java; classes and methods are used instead. Thus programmers are constrained to use a "pure" object-orientated style, rather than mixing the procedural and object oriented approaches.

Java provides built-in support for multiple threads of execution. This features allows different parts of an application's code to be executed concurrently. In Java, threads are implemented as objects. The Java library provides methods that can be invoked to start or stop a thread, check on the status of a thread, and synchronize the operation of multiple threads.

The Java compiler follows the P-code approach. It does not generate machine code or assembly language for a particular target machine. Instead, the compiler generates bytecodes – a high level, machine independent code for a hypothetical machine. This hypothetical machine is implemented on each target computer by an interpreter and run-time system. Thus a JAVA applications can be run without modification and without recompiling on any computer for which Java interpreter exists. The JAVA compiler itself is written in JAVA Therefore compiler can also run or any machine with a JAVA interpreter.

The bytecode approach also allows easy integration of JAVA applications into the world wide web. When a Java compatible Web browser is used to view the page, the code for the applet is downloaded and executed by the browser. The applet can then perform animation, play sound and generally interact with the user in real time.

The Java virtual machine supports a standard set of primitive data types :

1-, 2-, -4, and 8 byte integers, single and double – precision floating point numbers, and 16-bit character code. These data representations are independent of the architecture of the target machine. The interpreter is responsible for emulating these data types using the hardware.

A bytecode instruction on the JAVA Virtual Machine consists of a 1-byte opcode followed by zero or more operands. Many opcodes require no explicit operands in the instruction; instead, they take their operand values from a stack.

There are also single bytecode instructions that perform higher-level operations. For example, one instruction allocates a new array of a particular type. Other instruction tests whether an object is an instance of particular class. There are four instructions that can be used to invoke a method on an object. Another group of instructions is used to manipulate fields within an object.

The automatic garbage collection system used to manage memory runs as a low-priority background thread.

YACC (yet Another Compiler- Compiler) is a parser generator that is available on Unix systems. YACC has been used in the production of compilers for Pascal, RATFOR, APL, C and many other programming languages. It has also seen been used for several less conventional applications, including a typesetting language and a document retrieval system. In this section we give brief descriptions of YACC and LEX, the scanner generator that is related to YACC.

A lexical scanner must be supplied for use with YACC, This scanner is called by the parser whenever a new input token is needed. It returns an integer that identifies the type of token found. The scanner may also make entries in a symbol table for identifiers that are processed.

LEX is a scanner generator that can be used to create scanners of the type required by YACC. A portion of an input specification for LEX is shown in fig. Given below.

```
 .
  .
  .
“ ″      ;/*ignore blanks */
let     return (LET);
“ * ″   return (MUL);
“=″     return (ASSIGN);

[a – zA – z]
 [a – z A – Z 0 – 9]* {make entries in tables; return (ID)},
```

Each entry in the left hand column is a pattern to be matched against input stream. When a pattern is matched the corresponding action routine in the right-hand column is invoked. These action routine are written in the programming language C.

In the example shown above the first pattern has no associated action; the effect of this is to delete blanks as the input is scanned. The actions for next three patterns simply return a token type.

The token LET for the keyword let, MUL for the operator * and ASSIGN for the operator =. Internal representations of LET, MUL and the other tokens are integers. The fifth pattern specifies the form of identifies to be recognized. The first character must be in the range a-z or A-Z. This may be followed by any number of characters in the ranges a-z, A-Z or 0-9. The * in this pattern indicates that an arbitrary number of repetitions of preceding item are allowed. The action routine for this pattern makes entries in the appropriate tables to describe the identifier found and then return the token type ID.

The YACC parser generator accepts as input a grammar for the language being compiled and a set of actions corresponding to rules of the grammar. A portion of such an input specification appears in the fig given below.

```
% token ASSIGN  ID  LET  MUL…
       .
       .
       .
Statement   : LET ID ASSIGN expr
                    {…}
expr        : expr MUL expr
                    {$$ = build (MUL, $1, $3);}
expr        : ID
                    {…}
```

The first line shown is a declaration of the token types used. The other entries are rules of the grammar. The YACC parser calls the semantic routine associated with each rule as the corresponding language construct is recognized. Each routine may return a value by assigning it to the variable $$. Values returned by previous routines or by the scanner, may be referred to as$1, $2 etc. These variables designated the values returned for the components on the right-hand side of the corresponding rule reading from left to right.

It is sometimes useful to perform semantic processing as each part of a rule is recognize. YACC permits this by allowing semantic routines to be written in the middle of rule as well as at the end. The value returned by such a routine is available to any of the routines that appear later in the rule. It is also possible for user to define global variables that can be used by all of the semantic routines and by the lexical scanner.

The Parser generated by YACC use a bottom-up parsing method called LALR(1) which is a slightly restricted form of shift reduce parsing. The parsing produced by YACC have very good error detection properties. Error handling permits the reentry if the items in error or continuation of the input process after the erroneous entries are skipped.

University questions on this topic:

1.For the following Grammar construct the predictive parsing table and explain that step by step :

Grammar :
E → TE'
E' → +TE' | Є
T → FT'
T' → *FT' | Є
F → (E) | id
CMPN May 04, May 05, Dec 06 (10M)

2.Distinguish between Top-down and Bottom-up parsing Explain the LALR parser.
CMPN May 04 (10M)

3. Explain the phases of compiler with the help of example. **CMPN May 04 (10M)**
4. Distinguish between Parse tree and Syntax tree. Explain syntax directed translation. **CMPN May 04 (10M)**
5. Illustrate the various phases of compiler with respect to following statement :
   *position = initial + rate * 60 ;* **CMPN Dec 04, Dec 06 (10M)**
6. Define the finite state automata. What is their role in compiler theory explain in detail. **CMPN Dec 04 (10M), Dec 06 (04M**)
7. Explain LL (1) Parser with the help of example. **CMPN Dec 04, Dec 06 (10M)**
8. Define the following terms with the help of suitable example. **CMPN Dec 04 (10M)**
   - Parse tree
   - Syntax tree
   - Handle
   - Sentential form.
   - Context free grammar.

9. What is regular expression. Give regular expression which denotes all strings whose length is not two. Draw NFA for the same ? **CMPN May 05 (04M)**
10. What are various error recovery techniques used by compiler ? **CMPN May 05 (04M)**
11. Explain meaning of ambiguous Grammar, Left recursive Grammar and left factoring of a grammar with example. Give Non-ambiguous grammar for following pattern of a's, b's and c. WcW^{-1} where 'w' is any string of a's and b's and W^{-1} respresent recurse of string W. eg. aabab c babaa **CMPN May 05 (12M)**
12. Discuss various errors detected in each phase of compiler. **CMPN May 05 (08M)**
13. Give LEX and YACC specifications for program that translate your simple C code having FOR loop as below into your favorite assembly language. **CMPN Dec 05 (10M)**

```
begin
for i := 1 to n do
      for j := 1 to n do
       c[i,j] := 0
for i: = 1 to n do        .
      for j := 1 to n do
          for k := 1 to n do
           c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

14. For regular expression (0+1)* 01. Construct an NFA for this expression and convert this NFA to DFA. **CMPN Dec 05 (10M)**
15. Write notes on:

(i) Syntax directed translation (ii) Code optimization. **CMPN May 05 (10M)**

16. What are the various phases of compiler? Explain. **CMPN Dec 05 (10M)**
17. Explain Recursive-Descent parser as push down automata & Error recovery in a top down parser. **CMPN Dec 05 (10M)**
18. Describe the various forms of intermediate code used by compilers. **IT May 04, Dec 05 CMPN May 05 , May 06 , Dec 06 (10M)**
19. Consider the regular expression (a | b)* abb. Construct the NFA for this expression and convert this NFA to minimized DFA. **CMPN May 06 (10M)**
20. Explain Operator precedence parser with suitable example. **CMPN May 06, Dec 06 (10M)**

21. Discuss various errors detected in each phase of compiler. **CMPN May 06 (05 M), IT Dec 05 (10M)**

22. Write short notes on: **CMPN May 06 (12M)**
    (i) LEX and YACC. (ii) Recursive descent parser

23. What are various error recovery techniques used by compiler? **CMPN Dec 06 (04M)**

24. Discuss the loop optimization techniques with the help of suitable examples. **CMPN Dec 06 (10M)**

25. Explain syntax directed translation. Give Syntax directed definition to translate infix Expressions to Postfix Expressions. **CMPN Dec 06 (10M)**

26. State the reasons for the assembler to be multipass program. What are the different types of errors detected by a compiler ? Indicate which phase of the compiler identifies them with examples. **IT May 04 (10M)**

27. What is Parsing ? Differentiate top-down vs. bottom-up parsing method **IT May 04 (08M)**

28. What is Compiler ? Draw and explain the block diagram for structure of a general compiler. **IT Dec 04 (10M)**

29. What is regular expression ? Give regular expression which denotes all strings whose length is not two. Draw NFA for the same. **IT May 04 (10M)**

30. Write short notes on : Code Optimization in Compilers. **IT Dec 04 (07M), Dec 05 (10M)**

31. Differentiate assembler, compiler and interpreter. **IT May 05, Dec 05 (04M)**

32. "In every compiler there is an integrated assembler" justify. **IT May 05 (06M)**

33. What is code optimization in compilers ? What is need of optimization? Explain various types of code optimization techniques in compilers. **IT May 05 (10M)**

34. Differentiate between syntax tree and parse tree. Also explain, what is ambiguous grammar. Assume some suitable grammar. **IT May 05 (10M)**

35. Explain shift reduce parsing in detail. **IT May 05 (06M)**

36. Write short note on: Cross compiler. **IT May 05 (10M)**

37. Explain the conversion of a following source program into machine equivalent program using all phases of compiler : **(25 m) Old Syllabus**

```
Program Series :
VAR
I, sum : = integer
BEGIN
SUM : = 0
For I : =  1 to 9 DO
```

```
Sum : = sum + I;
Writeln (Sum)
END
```

38. What are the phases of a compiler ? Give the working of each phase for the following statements :- **Old Syllabus** 1

```
int      a, b, c   = 1;
a  =    a*b  - 5*3/c;
```

39. What are the phases of a compiler. Show the output of each phase for the following statements : **Old Syllabus** 1

```
void  fun  (int a)
{
printf ("%d\n", a);
}
```

40. How are functions linked ? Explain with the following examples for printf

```
void A ( )
  {
  Printf ("Linking done here\n");
  }
```

Old Syllabus