SYLLABUS

UNIT I

Introduction to System Software and Machine Structure: System programs – Assembler, Interpreter, Operating system. Machine Structure – instruction set and addressing modes. **Assemblers:** Basic assembler functions, machine – dependent and machine independent assembler features. Assembler design – Two-pass assembler with overlay structure, one – pass assembler and multi-pass assembler.

UNIT II

Loaders and Linkers: Basic loader functions, machine – dependent and machine – independent loader features. Loader design – Linkage editors, dynamic linking and bootstrap loaders.

UNIT III

Source Program Analysis: Compilers – Analysis of the Source Program – Phases of a Compiler – Cousins of Compiler – Grouping of Phases – Compiler Construction Tools.

Lexical Analysis: Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens –A Language for Specifying Lexical Analyzer.

UNIT IV

Parsing: Role of Parser – Context free Grammars – Writing a Grammar – Predictive Parser – LR Parser. **Intermediate Code Generation:** Intermediate Languages – Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back Patching – Procedure Calls.

UNIT V

Basic Optimization: Constant-Expression Evaluation – Algebraic Simplifications and Reassociation– Copy Propagation – Common Sub-expression Elimination – Loop-Invariant Code Motion – Induction Variable Optimization.

Code Generation: Issues in the Design of Code Generator – The Target Machine – Runtime Storage management – Next-use Information – A simple Code Generator – DAG Representation of Basic Blocks – Peephole Optimization – Generating Code from DAGs.

TEXT BOOKS

1. Alfred Aho, V. Ravi Sethi, and D. Jeffery Ullman, “Compilers Principles, Techniques and Tools”, Addison-Wesley, 1988. (UNITs III, IV & V)

2. Leland L. Beck, “System Software – In Introduction to System Programming”, Addison-Wesley, 1990 (UNITs I & II - Chapters: 1, 2 & 3).

REFERENCES

1. Allen Holub, “Compiler Design in C”, Prentice-Hall of India, 1990.

2. Charles N. Fischer and Richard J. Leblanc, “Crafting a Compiler with C”, Benjamin Cummings, 1991.

3. Steven S. Muchnick, “Advanced Compiler Design Implementation”, Morgan Koffman,1997.

4. Damdhare, “Introduction to System Software”, McGraw Hill, 1986.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: LANGUAGE TRANSLATORS **SUBJECT CODE: CST52**

UNIT I

Introduction to System Software and Machine Structure: System programs – Assembler, Interpreter, Operating system. Machine Structure – instruction set and addressing modes. **Assemblers:** Basic assembler functions, machine – dependent and machine independent assembler features. Assembler design – Two-pass assembler with overlay structure, one – pass assembler and multi-pass assembler.

2 MARKS

1. What is language translator?

A language translator is a program that takes as input a program written in one language and produces as output a program in another language. In program translation, the translator performs another very important role, the error-detection. Any violation of the High Level Language (HLL) specification would be detected and reported to the programmers.

2. What are the roles of translators?

Important role of translator are:

1. Translating the HLL program input into an equivalent Machine Language (ML) program.
2. Providing diagnostic messages wherever the programmer violates specification of the HLL.

3. What are the types of translators?

- Compiler
- Assembler
- Interpreter
- Pre-prossessor

4. Define system software. (MAY 2012) (NOV 2012)

System software consists of variety of programs that supports the operation of the computer. This software makes it possible for the user to focus on the other problems to be solved without needing to know how the machine works internally.

Eg: operating system, assembler, and loader.

5. What are the basic components of the system software?

- Operating system
- Compiler
- Assembler
- Macro processor
- Loader or linker
- Debugger
- Text editor
- Database management
- Software engineering tools

6. Difference between System software and Application Software.

System software	Application Software
System programming on the basic machine architecture.	It is independent of the machine architecture.
Its basic aim is to make the programming environment easier.	This environment focuses on the problem of the end users.
Example: compiler, loader	**Example:** payroll, banking

7. Define Operating systems.

Operating system acts as an interface between the user of a computer and the computer hardware. The operating system is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs.

Eg: Windows, Linux, UNIX, Dos

8. Give some applications of operating system.

- to make the computer easier to use
- Muti-tasking ,Multi-Programming
- Parallel Processing
- Spooling
- Buffering
- to manage the resources in computer
- process management
- data and memory management
- to provide security to the user

9. Define Assembler? (MAY 2012)

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

10. Define interpreter. (NOV 2011) (NOV 2012)

- Interpreter is a set of programs which converts high level language program to machine language program line by line.
- It can immediately execute high-level programs. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time

  and test them quickly.
- BASIC and LISP are especially designed to be executed by an interpreter. In addition, page description languages, such as PostScript, use an interpreter.

11. State the difference between assembler and Interpreter? (NOV 2013)

Assembler	Interpreter
An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.	Interpreter is a set of programs which converts high level language program to machine language program line by line.
An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter.	BASIC and LISP are especially designed to be executed by an interpreter. In addition, page description languages, such as PostScript, use an interpreter.

12. How could literals be implemented in one pass assembler? (MAY 2013)

- Each literal operand is recognized during pass1, the assembler searches LITTAB for the specified literal name or value.
- If the literal is already present in the table, no action is needed; if it is not present, the literal is added to LITTAB.
- When Pass1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table.

13. What are the different Machine Architectures?

There are two versions of Simplified Instructional Computer (SIC) machines Architectures are

- Standard model (SIC)
- XE version (SIC/XE)

  (“XE” eXtra Equipment or eXtra Expensive).

14. List the SIC machine architecture fields?

SIC consists of

- Memory
- Registers
- Data Formats
- Instruction Formats
- Addressing Modes
- Instruction Set
- Input and Output

15. What are the different registers in SIC Architecture?

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; jump to subroutine (JSUB) instruction stores the return address in this register
PC	8	Program Counter (PC); contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

16. What are the types of Addressing modes in SIC?

- Direct addressing mode
- Indexed addressing mode

Mode	Indication	Target address calculation
Direct	X=0	TA=address
Indexed	X=1	TA=address+(X)

17. Define Instruction set. (MAY 2013)

An **instruction set**, or **instruction set architecture** (**ISA**), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.

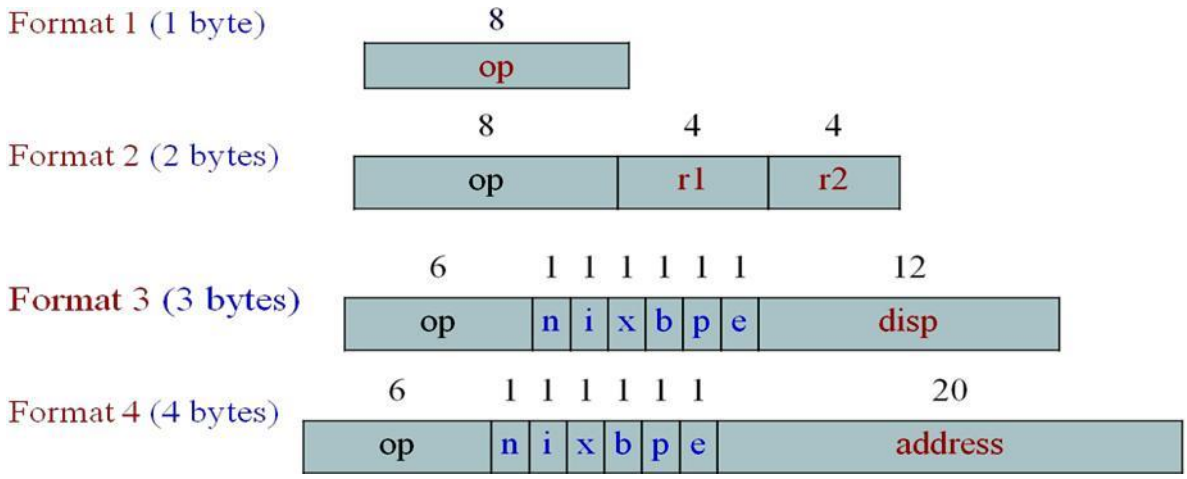
18. What are the types of Instruction set in SIC?

- Load and store registers: LDA, LDX, STA, STX, etc.
- Integer arithmetic Instructions: ADD, SUB, MUL, DIV, etc.
- Comparison Instructions: COMP
- COMP compares the value in register A with a word in memory, this instruction sets a Condition Code (CC) to indicate the result (<, =,>).
- Conditional jump instructions: JLT, JEQ, JGT
- Subroutine linkage: Jumps to the Subroutine (JSUB), RSUB

19. What are the different registers in SIC/XE Architecture?

Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator which is 48 bits

20. What are the different instruction formats in SIC/XE?



21. What are the types of addressing modes in SIC/XE?

- Base relative addressing mode
- Program Counter (PC) relative addressing mode
- direct addressing mode
- Indexed addressing mode
- immediate addressing mode
- indirect addressing mode

22. Differences between base relative and program counter relative addressing used in SIC/XE.

Mode	Indication	Target address calculation
Base relative	b=1,p=0	TA=(B)+ disp ($0 \leq disp \leq 4095$)
Program Counter (PC) relative	b=0,p=1	TA=(PC)+ disp ($-2048 \leq disp \leq 2047$)

23. Define indirect addressing

- In indirect addressing mode the value i=0, n=1:
- The word at the TA is fetched
- Value in this word is taken as the address of the operand value

Eg: ADD R5, [600]

Here the second operand is given in indirect addressing mode. First the word in memory location 600 is fetched and which will give the address of the operand.

24. Define immediate addressing.

- In this addressing mode the operand value is given directly. There is no need to refer memory.
- The immediate addressing is indicated by the prefix '#'.
  In case i = 1, n = 0 the address itself is the operand, no memory reference

Eg: ADD #5

In this instruction one operand is in accumulator and the second operand is an immediate value the value 5 is directly added with the accumulator content and the result is stored in accumulator.

25. List the instruction sets in SIC/XE?

- Load and store the new registers: LDB, STB, etc.
- Floating-point arithmetic operations: ADDF, SUBF, MULF, DIVF
- Register MOve: RMO
- Register-to-register arithmetic operations: ADDR, SUBR, MULR, DIVR
- Supervisor call(SVC)

26. Write the program for BETA = ALPHA + INCR -1 and DELTA=GAMMA + INCR -1 using SIC instructions.

```
            LDA     ALPHA   LOAD ALPHA INTO REGISTER A
            ADD     INCR    ADD THE VALUE OF INCR
            SUB     ONE     SUBTRACT 1
            STA     BETA    STORE IN BETA
            LDA     GAMMA   LOAD GAMMA INTO REGISTER A
            ADD     INCR    ADD THE VALUE OF INCR
            SUB     ONE     SUBTRACT 1
            STA     DELTA   STORE IN DELTA
            ….. ……..
ONE         WORD    1       ONE WORD CONSTANT
ALPHA       RESW    1       ONE WORD VARIABLES
BETA        RESW    1
GAMMA       RESW    1
DELTA       RESW    1
INCR        RESW    1
```

---

27. Write the program for BETA = ALPHA + INCR+1 and DELTA=GAMMA + INCR -1 using SIC /XE instructions.

```
            LDS     INCR    LOAD VALUE OF INCR INTO REGISTER S
            LDA     ALPHA   LOAD ALPHA INTO REGISTER A
            ADDR    S, A    ADD THE VALUE OF INCR
            SUB     #1      SUBTRACT 1
            STA     BETA    STORE IN BETA
            LDA     GAMMA   LOAD GAMMA INTO REGISTER A
            ADDR    S, A    ADD THE VALUE OF INCR
            SUB     #1      SUBTRACT 1
            STA     DELTA   STORE IN DELTA
            ….. ……..
ALPHA       RESW    1       ONE WORD VARIABLES
BETA        RESW    1
GAMMA       RESW    1
DELTA       RESW    1
INCR        RESW    1
```

28. What are the fields available in an assembly language instruction? (NOV 2011)

The four fields available in an assembly language instruction are

- Label
- Opcode
- Operand
- Comments

29. What are the different assembler directives? (NOV 2012)

START - specify program name and starting address of the program.

END - indicate the end of the source program.

BYTE - generate character or hexadecimal

WORD - constants. Generate one word integer constant.

RESB - Reserve the indicated no. of bytes for a data

RESW - area. Reserve the indicated no. of words for a data area.

30. What are the basic assembler functions?

1. Convert mnemonic operation codes to their machine language equivalents.
2. Convert symbolic operands to their equivalent machine addresses.
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into internal machine representations.
5. Write the object program and the assembly listing.

31. What are the functions of two pass assembler?

Functions of Two Pass Assembler

Pass 1 - define symbols (assign addresses)

- Assign addresses to all statements in the program.
- Save the values assigned to all labels for use in Pass 2.
- Perform some processing assembler directives.

Pass 2 - assemble instructions and generate object program

- Assemble instructions
- Generate data values defined by BYTE, WORD, etc.
- Perform some processing assembler directives not done in Pass 1.
- Write the object program and the assembly listing.

32. What is the format of the Object Program generated by the Assembler?

It contains 3 types of records:

Header record:

Col. 1	H
Col. 2~7	Program name
Col. 8~13	Starting address of object program (hexadecimal)
Col. 14-19	Length of object program in bytes (hexadecimal)

Text record

Col.1	T
Col. 2~7	Starting address for object code in this record (hex)
Col. 8~9	Length of object code in this record in bytes (hex)
Col.10~69	Object code (69-10+1)/6=10 instructions

End record

Col.1	E
Col.2~7	Address of first executable instruction in object program (hex)

33. What is the use of OPTAB? (MAY 2012)

- The Operation code table (OPTAB) contains the mnemonic operation code and its machine language equivalent.
- Some assemblers it may also contain information about instruction format and length.
- OPTAB is usually organized as a hash table, with mnemonic operation code as the key.

34. What is the use of SYMTAB? (MAY 2012)

- Symbol table (SYMTAB) - is used to store values (addresses) assigned to labels.
- It includes the name and value for each symbol in the source program, together with flags to indicate error conditions.
- Sometimes it may contain details about the data area.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.

35. What are forward references?

It is a reference to a label that is defined later in a program.

Consider the statement

```
                          RETAD
10   1000     STL         R
. . . .
. . . .
80   1036     RETADR      RESW
```

The first instruction contains a forward reference RETADR. If we attempt to translate the program line by line, we will unable to process the statement in line10 because we do not know the address that will be assigned to RETADR .The address is assigned later in the program.

36. List the features of machine dependent Assembler. (NOV 2013)

The features of machine dependent Assembler is

1. Instruction formats
2. Addressing modes and
3. Program Relocation

37. Define relocatable program.

An object program that contains the information necessary to perform required modification in the object code depends on the starting location of the program during load time is known as relocatable program.

38. Define modification record and give its format.

In Modification record contains the information about the modification in the object code during program relocation.

The general format is

Col 1 M

Col 2-7 Starting location of the address field to be modified relative to the beginning of the program.

Col 8-9 Length of the address field to be modified in half bytes.

39. Define literals?

Literal is a constant operand which is used to write the value of it as a part of the instruction. This avoids having to define the constant elsewhere in the program and make up a label for it.

40. Define Literal pools.

All of the literal operands used in a program are gathered together into one or more literal pools. Normally literals are placed into pool at the end of the program.

41. What are the symbol defining statements generally used in assemblers?

- **'EQU'** - it allows the programmer to define symbols and specify their values directly. The general format is

  Symbol EQU value

- **'ORG'** - it is used to indirectly assign values to symbols. When this statement is encountered the assembler resets its location counter to the specified value.

  The general format is

  ORG value

42. Differentiate absolute expression and relative expression.

- If the result of the expression is an absolute value (constant) then it is known as absolute expression.

  Eg: BUFEND – BUFFER

- If the result of the expression is relative to the beginning of the program then it is known as relative expression. Label on instructions and data areas and references to the location counter values are relative terms.

  Eg: BUFEND + BUFFER

43. Define Program blocks.

The program blocks refer to segments of code that are rearranged within a single object program unit.

44. What are the types of program blocks?

The three blocks are

1. Default – it contains the executable instructions of the program.
2. CDATA – it contains all data areas that are a few words or less in length.
3. CBLKS – it contains all data areas that consist of larger block memory.

45. Define control section.

A control section is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or logical subdivisions of a program. The programmer can assemble, load and manipulate each of these control sections separately. The resulting flexibility is a major benefit of these control sections.

46. What is program linking?

- When control sections from logically related parts of a program, it is necessary to provide means for linking them together.
- For example, instructions in one control section might need to refer to instructions or data located in another section.

47. What is meant by external references?

In assembler where any other control section will be located at execution time. Such references between controls are called external references. The assembler generates information for each external reference that will allow the loader to perform the required linking.

48. What are the types of external symbols?

The two assembler directives to identify such references

1. External Definition (EXTDEF) names external symbols that are defined in a particular control section and may be used in other sections.
2. External References (EXTREF) names external symbols that are referred in a particular control section and defined in another control section.

49. Write the Format for Define and Refer record?

DERFINE RECORD

Col 1	D
Col 2-7	Name of external symbol defined in this control section
Col 8-13	Relative address of symbol within this control section
Col 14-73	Repeat information in col 2-13 for other external symbol

REFER RECORD

Col 1	R
Col 2-7	Name of external symbol defined in this control section
Col 8-73	Name of other external reference symbol

50. What is the use of load and go assembler?

- One pass assembler that generates their object code in memory for immediate execution is known as load and go assembler. No object programmer is written out and no loader is needed.
- It is useful in a system that is oriented toward program development and testing.

51. What is one pass assembler?

- One-pass assembler that generates their object code in memory for immediate execution.
- One-pass assemblers are used when it is necessary or desirable to avoid a second pass over the source program the external storage for the intermediate file between two passes is slow or is inconvenient to use.

Main problem: forward references to both data and instructions. One simple way to eliminate this problem: require that all areas be defined before they are referenced. It is possible, although inconvenient, to do so for data items.

- Forward jump to instruction items cannot be easily eliminated.
- Sample Program for a One-Pass Assembler.

52. What is multi-pass assembler?

Prohibiting forward references in symbol definition:

- This restriction is not a serious inconvenience.
- Forward references tend to create difficulty for a person reading the program.

Allowing forward references

- To provide more flexibility.

11 MARKS

1. Explain the system software and Machine structures? (11 marks)

System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer.

Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems and software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

Types of software:

1. Application software
2. System software

One characteristic in which most system software differs from application software is machine dependency.

- System software supports operation and use of computer.
- Application software provides solution to a problem.

Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly, Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

There are aspects of system software that do not directly depend upon the type of computing system, general design and logic of an assembler, general design and logic of a compiler and code optimization techniques, which are independent of target machines. Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.

Components of system software

The language translator and the operating system are themselves programs. Their function is to get the user's program, which is written in a programming language, to run on the computer system. The programs which help in the execution of a user program are called **System Programs (SPs).** The collection of such SPs is the "system software" of a particular computer system. An identical argument holds in the case of a computer system. By writing a program in a higher level programming language, a programmer obviates the need to acquire totally new skills merely in order to run his program. The compiler performs the task of making his program understandable to the CPU.

Two fundamental aspects of the system are

- Making available new/better facilities
- Achieving efficient performance

The basic components of the system software are,

1. Interpreter
2. Assembler
3. Macro processing
4. Linking loader
5. Absolute loader
6. Operating system
7. Compiler
8. Debugging aids
9. Text editors
10. Utilities

Interpreter

This is also a translator converts the high level language into low level language. It converts the program line by line at a time. It analyses the source program statement by statement and it carries out the actions implied by each statement.

Assembler

It is a type of translator that converts assemble level language into machine level language. An assembler is used to convert the given mnemonics into numeric and converts them to executable or command files. Output is an object program plus program information that enables the loader to prepare the object program for execution.

Preprocessor

It is a translator that coverts one high level language into another high level language.

Macro processor

A macro call is an abbreviation for some codes. A macro definition is a sequence of code that has a name. a macro processor is a program that substitutes and specifies macro definition for macro calls.

Loader

Loader loads the object program and prepare for its execution. Loader schemes are absolute, relocating and direct linking. In general loader must load, relocate and link the object program.

Operating system

An operating system (OS) is a software program that manages the hardware and software resources of a computer. The OS performs basic tasks, such as controlling and allocating memory, prioritizing the processing of instructions, controlling input and output devices facilitating networking, and managing files.

Compiler

Compiler is a computer program (or set of programs) that translates text written in a computer language (the source language) into another computer language (the target language). The original sequence is usually called the source code and the output called object code. Commonly the output has a form suitable for processing by other programs (e.g., a linker), but it may be a human readable text file.

System Software and Architecture:

- Machine dependency of system software
- System programs are intended to support the operation and use of the computer. Machine architecture differs in:
    - Machine code
    - Memory
    - Instruction formats
    - Addressing modes
    - Registers
- Machine independency of system software

  General design and logic is basically the same:
    - Code optimization
    - Subprogram linking

2. Write a short note on Assembler? (5 marks)

- An **Assembler** translates a program written in an assembly language to it machine language equivalent.
- An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.
- An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.
- An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter.
- An assembler is a program that accepts an assembly language program as input and produces its machine language equivalent along with information for the loader.
- Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.
- Assembly language uses a mnemonic to represent each low-level machine instruction or operation. Typical operations require one or more operands in order to form a complete instruction, and most assemblers can therefore take labels, symbols and expressions as operands to represent addresses and other constants, freeing the programmer from tedious manual calculations.

- **Macro assemblers** include a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.

Number of passes

There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
- Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

High-level assemblers

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures.
- High-level procedure/function declarations and invocations.
- High-level abstract data types, including structures/records, unions, classes, and sets.
- Sophisticated macro processing.
- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance.

Basic elements

There is a large degree of diversity in the way the authors of assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation.

A typical assembly language consists of 3 types of instruction statements that are used to define program operations:

- Opcode mnemonics
- Data directives
- Assembly directives

Typical applications

- Assembly language is typically used in a system's boot code, (BIOS on IBM-compatible PC systems).
- computer cartridge games
- Microcontrollers (automobiles, industrial plants...)
- telecommunication equipment
- device drivers
- Very fast and compact but processor-specific.

3. Write a short note on Interpreter? (5 marks)

- ***Interpreter*** is also a translator converts the high level language into low level language. It converts the program line by line at a time.
- It analyses the source program statement by statement and it carries out the actions implied by each statement.
- It can immediately execute high-level programs. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly.
- BASIC and LISP are especially designed to be executed by an interpreter. In addition, page description languages, such as PostScript, use an interpreter.

An ***interpreter*** is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously batch-compiling them into machine language. An interpreter generally uses one of the following strategies for program execution:

1. parse the source code and perform its behavior directly
2. translate source code into some efficient intermediate representation and immediately execute
3. explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

Early versions of the Lisp programming language and Dartmouth BASIC would be examples of the first type. Perl, Python, MATLAB, and Ruby are examples of the second, while UCSD Pascal is an example of the third type. Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and compiler. Some systems, such as Smalltalk, contemporary versions of BASIC and Java.

While interpretation and compilation are the two main means by which programming languages are implemented, they are not mutually exclusive, as most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" signify that the canonical implementation of that language is an interpreter or a compiler, respectively. A high level language is ideally an abstraction independent of particular implementations.

Compiler

Pros

- Less space
- Fast execution

Cons

- Slow processing
- Partly Solved (Separate compilation)
- Debugging
- Improved thru IDEs

Interpreter

Pro

- Easy debugging
- Fast Development

Cons

- Not for large projects
- Exceptions: Perl, Python
- Requires more space
- Slower execution
- Interpreter in memory all the time

4. Write a short note on Operating system? (6 marks)

- An **operating system (OS)** is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is an essential component of the system software in a computer system. Application programs usually require an operating system to function.
- **Operating system** acts as an interface between the user of a computer and the computer hardware.
- The operating system is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs.
- **Eg: Windows, Linux, UNIX, Dos**

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware,[1][2] although the application code is usually executed directly by the hardware and will frequently make a system call to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a computer from cellular phones and video game consoles to supercomputers and web servers.

Examples of popular modern operating systems include Android, BSD, iOS, Linux, OS X, QNX, Microsoft Windows, Windows Phone, and IBM z/OS.

Types of operating systems

Real-time

A real-time operating system is a multitasking operating system that aims at executing real-time

applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or time-sharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

Multi-user

A multi-user operating system allows multiple users to access a computer system at the same time. Time-sharing systems and Internet servers can be classified as multi-user systems as they enable multiple-user access to a computer through the sharing of time. Single-user operating systems have only one user but may allow multiple programs to run at the same time.

Multi-tasking vs. single-tasking

A multi-tasking operating system allows more than one program to be running at the same time, from the point of view of human time scales. A single-tasking system has only one running program. Multi-tasking can be of two types: pre-emptive and co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs. In 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions of both Windows NT and Win9x used pre-emptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

Distributed System

A distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

Templated

In an OS, distributed and cloud computing context, templating refers to creating a single virtual machine image as a guest operating system, then saving it as a tool for multiple running virtual machines. The technique is used both in virtualization and cloud computing management, and is common in large server warehouses.

Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

Components of Operating System

The components of an operating system all exist in order to make the different parts of a computer work together. All user software needs to go through the operating system in order to use any of the hardware, whether it is as simple as a mouse or keyboard or as complex as an Internet component.

- Kernel
- Program execution
- Interrupts
- Protected mode and Supervisor mode
- Memory management
- Virtual memory
- Multitasking
- Disk access and file systems
- Device drivers
- Networking
- Security
- User interface

Applications of Operating System

- Muti-tasking
- Multi-Programming
- Parallel Processing
- Spooling
- Buffering
- process management
- data and memory management
- to provide security to the user

5. Explain in detail about SIC Machine Architecture? (11 marks) (NOV 2011)(MAY 2013)(NOV 2013)

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines.

There are two versions of SIC are

- Standard model (SIC)
- XE version (SIC/XE) ("XE" eXtra Equipment or eXtra Expensive).

SIC Machine Architecture:

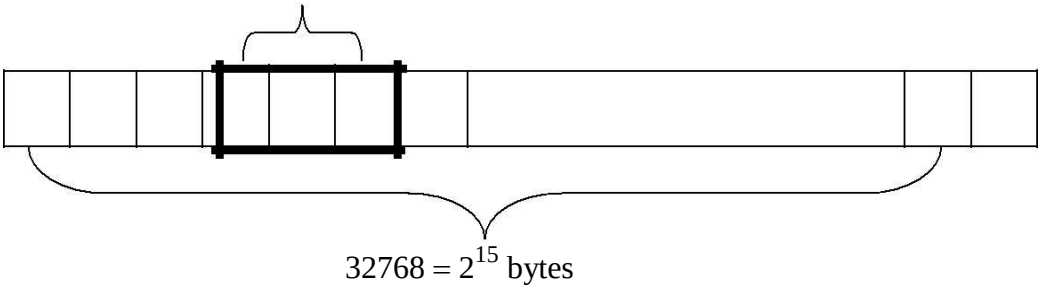
It consists of

1. Memory
2. Registers
3. Data Formats
4. Instruction Formats
5. Addressing Modes
6. Instruction Set
7. Input and Output

MEMORY

- It consists of 8-bit bytes.
- 3 consecutive bytes forms word (24 bits).
- All addresses on SIC are byte addresses.
- Words are addressed by the location of their lowest numbered byte
- 32,768 (2^{15}) bytes in the computer memory.

A word (3 bytes or 24 bits)



$32768 = 2^{15}$ bytes

REGISTERS

- There are 5 registers.
- Each register is 24 bits in length.

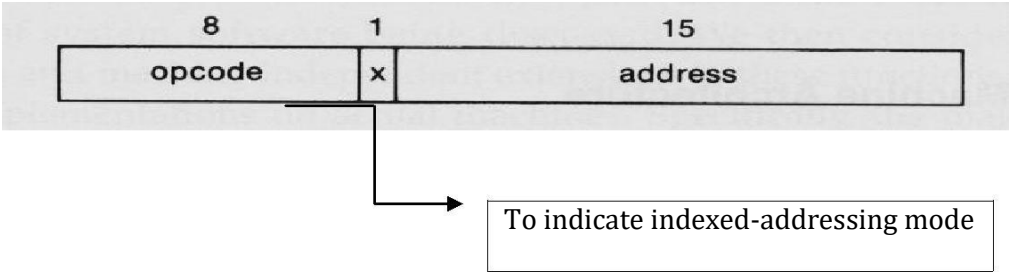
Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; jump to subroutine (JSUB) instruction stores the return address in this register
PC	8	Program Counter (PC); contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

DATA FORMATS

- Integers are stored as 24-bit binary numbers
- Characters :8-bit ASCII codes
- 2's complement for negative values
- There is no Floating-point numbers on Standard version SIC.

INSTRUCTION FORMATS

- All machine instructions on the standard SIC have 24-bit format.



ADDRESSING MODES

The two types of addressing modes are

1. Direct addressing mode
2. Indexed addressing mode
   - Set X bit to 0 or 1
   - Target address (TA) is calculated.

Mode	Indication	Target address calculation
Direct	X=0	TA=address
Indexed	X=1	TA=address+(X)

(X): Contents of a register or a memory location

INSTRUCTION SET

- Load and store registers: LDA, LDX, STA, STX, etc.
- Integer arithmetic Instructions: ADD, SUB, MUL, DIV, etc.
- All arithmetic operations involve register **A** and a word in memory, with the result being left in the register.
- Comparison Instructions : COMP
- COMP compares the value in register A with a word in memory, this instruction sets a condition code (CC) to indicate the result (<, =,>).
- Conditional jump instructions: JLT, JEQ, JGT
  - These instructions test the setting of CC and jump accordingly.
- Subroutine linkage: JSUB, RSUB
  - JSUB jumps to the subroutine, placing the return address in register L.
  - RSUB returns by jumping to the address contained in register L.

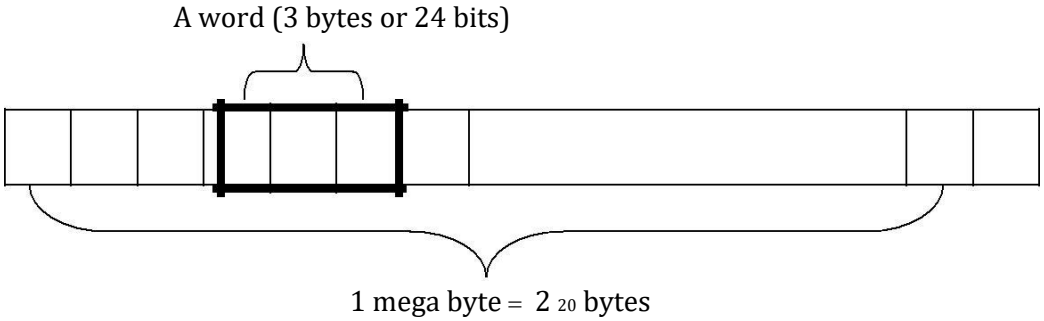
INPUT/OUTPUT

- Each IO device is assigned a unique 8-bit code.
- One byte at a time to or from the rightmost 8 bits of register A.
- Three instructions:
  - Test device (TD)
    - Test whether the device is ready to send/receive
    - Test result is set in CC
  - Read data (RD): read one byte from the device to register A.
  - Write data (WD): write one byte from register A to the device.

6. Explain in detail about SIC/XE Machine Architecture? (11 marks)

Memory

- Memory is same as SIC standard version.
- Maximum memory available on a SIC/XE system is 1 megabyte



Registers

- Additional registers are provided by SIC/XE.

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; jump to subroutine (JSUB) instruction stores the return address in this register
PC	8	Program Counter (PC); contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

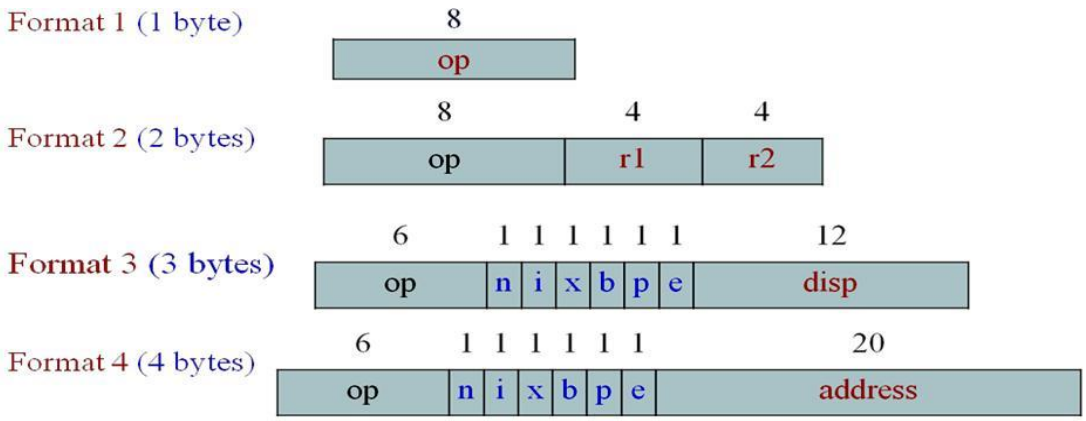
Data Formats

- There is a 48-bit floating-point data type.
- Sign bit 0 (+ve) or 1 (-ve).
- exponent is an unsigned binary number between 0 and 2047.
- fraction is a value between 0 and 1.

1	11	36
S	**Exponent**	**fraction**

Instruction Formats

- Use relative addressing.
- Extend the address field to 20 bits.
- Instruction that reference memory uses.
- Formats 1 and 2 do not reference memory at all.
- Bit e distinguishes between format 3 and 4.



Addressing Modes

- Base relative addressing mode
- Program Counter (PC) relative addressing mode
- direct addressing mode
- Indexed addressing mode
- immediate addressing mode
- indirect addressing mode

Base Relative addressing mode

- The displacement field in format 3 instruction is interpreted as a 12 bit unsigned integer.
- Ex: 1056 STX LENGTH

Mode	Indication	Target address calculation
Base relative	b=1,p=0	**TA = (B)+ disp** (0 ≤ disp ≤ 4095)
Program Counter (PC) relative	b=0,p=1	**TA =(PC)+ disp** (-2048 ≤ disp ≤ 2047)

Program counter Relative Addressing mode

- The displacement field in format 3 instruction is interpreted as a 12 bit signed integer.
- Negative values 2's complement.
- Ex: 0000 STL RETADDR

Direct addressing mode

- In direct addressing for formats 3 and 4 if b=0, p=0, x=0
- Ex: LDA LENGTH
- **TA = Address field (0 ≤ disp ≤ 4095).**

Indexed addressing mode

- Set bit x=1, b=1, p=0 value to be added to the value stored at the register x to obtain real address of the operand.
- STCH BUFFER , X
- **TA = (B) + (X) + disp**

Immediate addressing mode

- Set bit i=1, n=0: immediate addressing
- TA is used as the operand value, no memory reference
- Ex: LDA #9
- **TA = operand value = disp**

Indirect Addressing mode

- Set bit i=0, n=1, x=0
- he word at the TA is fetched value in this word is taken as the address of the operand value
- Ex: 002A J @ RETADDR

Instruction Set

- Load and store the new registers: LDB, STB, etc.
- Floating-point arithmetic operations- ADDF, SUBF, MULF, DIVF
- Register move: RMO
- Register-to-register arithmetic operations- ADDR, SUBR, MULR, DIVR
- Supervisor call: SVC-for generating an interrupt.

Input and output

- There are I/O channels that can be used to perform input and output while the CPU is executing other instructions.
- The instructions SIO, TIO, and HIO are used to start test and halt the operation of I/O channels.

7. Write a SIC and SIC/XE assembler programs? (11 marks) (MAY 2013)

Data Movement Operation:

ALPHA=5, C1=Z using SIC instructions

```
            LDA     FIVE    LOAD CONSTANT 5 INTO REGISTER A
            STA     ALPHA   STORE IN ALPHA
            LDCH    CHARZ   LOAD CHARACTER 'Z' INTO REGISTER A
            STCH    C1      STORE IN CHARACTER VARIABLE C1
            …….
ALPHA       RESW    1       ONE WORD VARIABLE
FIVE        WORD    5       ONE WORD CONSTANT
CHARZ       BYTE    C'Z'    ONE BYTE CONSTANT
C1          RESB    1       ONE BYTEVARIABLE
```

ALPHA=5, C1=Z using SIC/XE instructions

```
            LDA     #5      LOAD VALUE 5 INTO REGISTER A
            STA     ALPHA   STORE IN ALPHA
            LDA     #90     LOAD ASCII CODE FOR 'Z' INTO REGISTER A
            STCH    C1      STORE IN CHARACTER VARIABLE C1
            ……
ALPHA       RESW    1       ONE WORD VARIABLE
C1          RESB    1       ONE BYTEVARIABLE
```

Arithmetic Operation:

BETA = ALPHA + INCR+1 and DELTA=GAMMA + INCR -1 using SIC instructions.

```
                LDA     ALPHA   LOAD ALPHA INTO REGISTER A
                ADD     INCR    ADD THE VALUE OF INCR
                SUB     ONE     SUBTRACT 1
                STA     BETA    STORE IN BETA
                LDA     GAMMA   LOAD GAMMA INTO REGISTER A
                ADD     INCR    ADD THE VALUE OF INCR
                SUB     ONE     SUBTRACT 1
                STA     DELTA   STORE IN DELTA
                ..... ........
ONE             WORD    1       ONE WORD CONSTANT
ALPHA           RESW    1       ONE WORD VARIABLES
BETA            RESW    1
GAMMA           RESW    1
DELTA           RESW    1
INCR            RESW    1
```

BETA = ALPHA + INCR+1 and DELTA=GAMMA + INCR -1 using SIC /XE instructions.

```
                LDS     INCR    LOAD VALUE OF INCR INTO REGISTER S
                LDA     ALPHA   LOAD ALPHA INTO REGISTER A
                ADDR    S, A    ADD THE VALUE OF INCR
                SUB     #1      SUBTRACT 1
                STA     BETA    STORE IN BETA
                LDA     GAMMA   LOAD GAMMA INTO REGISTER A
                ADDR    S, A    ADD THE VALUE OF INCR
                SUB     #1      SUBTRACT 1
                STA     DELTA   STORE IN DELTA
                ..... ........
ALPHA           RESW    1       ONE WORD VARIABLES
BETA            RESW    1
GAMMA           RESW    1
DELTA           RESW    1
INCR            RESW    1
```

8. Explain in detail about basic assembler functions? (11 marks) (NOV 2013)

- An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.
- An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter.

Basic Assembler Functions:

The basic assembler functions are:

- Translating mnemonic operation codes to their machine language equivalents.
- Assigning machine addresses to symbolic labels.

Assembler Directives

The various basic assembler directives are

- START - specify program name and starting address of the program.
- END - indicate the end of the source program
- BYTE - generate character or hexadecimal constants.
- WORD - Generate one word integer constant.
- RESB - Reserve the indicated no. of bytes for a data area.
- RESW - Reserve the indicated no. of words for a data area.

Assembler's Functions

1. Convert mnemonic operation codes to their machine language equivalents.
2. Convert symbolic operands to their equivalent machine addresses.
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into internal machine representations.
5. Write the object program and the assembly listing

Functions of Two Pass Assembler

The assembled program will be loaded into memory for execution.

Pass 1 - define symbols (assign addresses)

- Assign addresses to all statements in the program.
- Save the values assigned to all labels for use in Pass 2.
- Perform some processing assembler directives.

Pass 2 - assemble instructions and generate object program

- Assemble instructions.
- Generate data values defined by BYTE, WORD, etc.
- Perform some processing assembler directives not done in Pass 1.
- Write the object program and the assembly listing.

The simple object program contains three types of records:

1. Header record
2. Text record and
3. End record.

- The header record contains the starting address and length.
- Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded.

- The end record marks the end of the object program and specifies the address where the execution is to begin.

Header record:

Col. 1	H
Col. 2~7	Program name
Col. 8~13	Starting address of object program (hexadecimal)
Col. 14-19	Length of object program in bytes (hexadecimal)

Text record

Col.1	T
Col. 2~7	Starting address for object code in this record (hex)
Col. 8~9	Length of object code in this record in bytes (hex)
Col.10~69	Object code (69-10+1)/6=10 instructions

End record

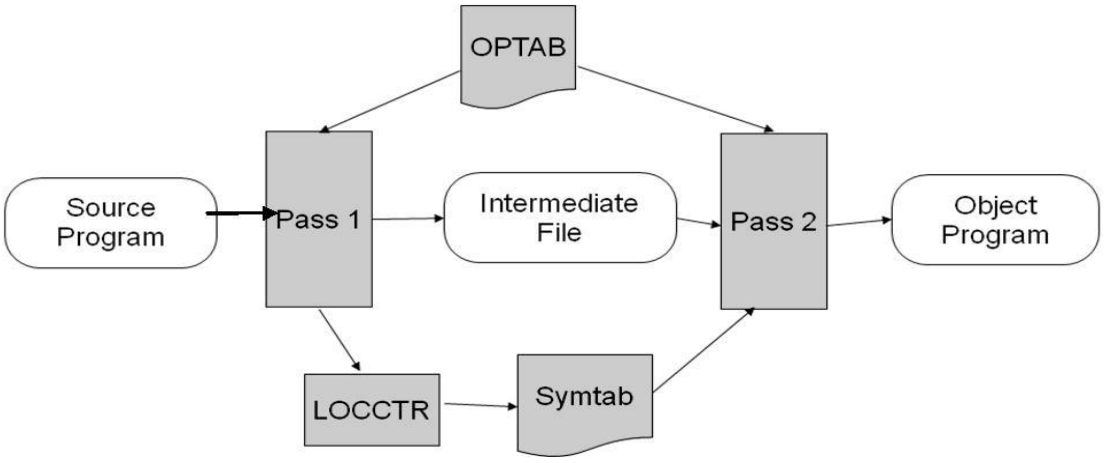
Col.1	E
Col.2~7	Address of first executable instruction in object program (hex)

9. Discuss about algorithms and data structure with an example? (11 marks)

Algorithms and Data structures:

The simple assembler uses two major internal data structures:

1. Operation Code Table (OPTAB)
2. Symbol Table (SYMTAB)



OPTAB (Operation Code Table):

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents.
- In more complex assemblers the table also contains information about instruction format and length.
- OPTAB is usually organized as a hash table, with mnemonic operation code as the key

- The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching.
- In pass 1 the OPTAB is used to look up and validate the operation code in the source program.

  In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2.
- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

SYMTAB (Symbol Table):

- SYMTAB is used to store values (addresses) assigned to labels.
- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions

LOCCTR:

- Location Counter (LOCCTR) is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction.
- Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

Pass 1

Algorithm: begin

read first input line

if OPCODE = 'START'

then begin

save #[Operand] as starting address

initialize LOCCTR to starting address

write line to intermediate file

read next

line **end**(if START)

else

initialize LOCCTR to

0 **while** OPCODE != 'END' **do**

begin

if this is not a comment line

then begin

if there is a symbol in the LABEL field

then begin

search SYMTAB for LABEL **if** found **then**

```
                    set error flag (duplicate
                    symbol) else {if symbol}
                search OPTAB for
                OPCODE if found then
                    add 3 (instr length) to LOCCTR
                else if OPCODE = 'WORD' then
                    add 3 to LOCCTR
                else if OPCODE = 'RESW' then
                    add 3 * #[OPERAND] to LOCCTR

            else if OPCODE = 'RESB' then
                    add #[OPERAND] to LOCCTR
            else if OPCODE = 'BYTE' then
                    begin
                        find length of constant in bytes
                        add length to LOCCTR
            end {if BYTE}
                    else
                        set error flag (invalid operation code)
                end (if not a comment)
            write line to intermediate file
            read next input line
        end { while not END} write
    last line to intermediate file
  Save (LOCCTR – starting address) as program length
end {Pass1}
```

- The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.
- If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.
- If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.
- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes.

- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

Pass 2 Algorithm:

```
begin
    read first input line {from intermediate file}
    if OPCODE = 'START' then
        begin
            write listing line
            read next input line
        end {if START}
    write Header record to object
    program initialize first Text record
    while OPCODE != 'END'
        do begin
            if this is not comment line
                then begin
                    search OPTAB for OPCODE
                if found then
                  begin
                    if there is a symbol in OPERAND field then
                      begin
                        search SYMTAB for OPERAND
                        if found then
                    begin
                        store symbol value as operand address
                        set error flag (undefined symbol)
                    end
                end {if symbol}
            else
                store 0 as operand address
            assemble the object code instruction
        end {if symbol}
    else if OPCODE = 'BYTE' or 'WORD" then
        convert constant to object code
    if object code doesn't fit into current Text record
        then begin
            Write text record to object
            code initialize new Text record
        end

        add object code to Text
    record end {if not comment}
write listing line
        read next input
```

line **end {while not END}**

write last Text record to object

program write End record to object

program write last listing line

end {Pass 2}

- The first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.
- If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.
- If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code (for example, for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written

EXAMPLE: SIC –Standard Version

The Source program is **P = X + Y + 2XY**

LOCCTR	**LABEL**	**OPCO DE**	**OPERAND**	**OBJECT CODE**
2000	**SAMPLE**	**START**	**2000**	
2000		LDA	X	**002018**
2003		ADD	Y	**18201B**
2006		STA	TEMP	**C0201E**
2009		LDA	X	**002018**
200C		MUL	Y	**20201B**
200F		MUL	TWO	**202024**
2012		ADD	TEMP	**18201E**
2015		STA	P	**0C2021**
2018	X	RESW	1	
201B	Y	RESW	1	
201E	TEMP	RESW	1	
2021	P	RESW	1	
2024	TWO	WORD	2	**000002**
2027	**END**			

The object program of this source program:

H^SAMPLE^002000^000027

T^002000^**1B^**002018^18201B^0C201E^002018^20021B^202024^18201E^0C2021^000002

E^002000

SYMTAB

SYMBOL	VALUE
X	2018
Y	201B
TEMP	201E
P	2021
TWO	2024

OPTAB

MNEMONICS	OPERAND
LDA	00
ADD	18
STA	0C
MUL	20

10. Explain in detail about machine dependent assembler features? (11 marks) (MAY 2012)

Consider the design and implementation of an assembler for the more complex XE version of SIC.

- instruction formats
- addressing modes
- program relocation

(i)Instruction Format:

Instructions can be:

- Instructions involving register to register
- Instructions with one operand in memory, the other in Accumulator
- Extended instruction format

Format 1:

Length = 1 byte

8 bits

OPCODE

Format 2:

Length = 2 bytes

It is used for register –to – register instruction

8bits 4 4

OPCODE	r1	r2

Format 3:

Length = 3 bytes

Register –memory instruction.

6 1 1 1 1 1 1 12

OPCODE	n	i	x	b	p	e	Displacement

Displacement can be calculated in two ways

- PC relative

- Base relative

PC –relative displacement Calculation

TA= [PC] + disp (-2048 ≤ displacement ≤ 2047)

BASE – relative displacement Calculation

TA= (B) + disp (0 ≤ disp ≤ 4095)

Format 4 :(Extended format) (if disp>4095)

Extended format of the instruction must be indicated by the prefix (+)

6	1	1	1	1	1	1	20
OPCODE	n	i	x	b	p	e	Address

Translation

1. Register Translations for the Instruction involving Register-Register addressing mode

- Register name (A, X, L, B, S, T, F, PC, SW) and their values (0, 1, 2, 3, 4, 5, 6, 8, 9).
- **During pass 1** the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes.
- **During pass2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2. Translation involving Register-Memory instructions:

- In SIC/XE machine there are four instruction formats and five addressing modes.
- For formats and addressing modes
- The instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction.
- One of the operand is always in a register and the other operand is in the memory.
- The addressing mode tells us the way in which the operand from the memory is to be fetched.

3. Address Translation

- Most register-memory instructions use program counter relative or base relative addressing
- Format 3: 12-bit disp field
  - Base-relative: 0~4095
  - PC-relative: -2048~2047
- Format 4: 20-bit address field

(ii) Addressing Modes:

- PC-relative or Base-relative addressing op m
- Indirect addressing op @ m
- Immediate addressing op #c
- Extended format + op m
- Index addressing op m , x

Program-Counter Relative:

- In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value.
- The range of displacement values are from 0 -2048. This displacement value is added to the current contents of the program counter to get the target address of the operand required by the instruction.

TA= [PC] + disp -2048 ≤ disp ≤ 2047

Base-Relative Addressing Mode:

- In this mode the base register is used to mention the displacement value.
- Therefore the target address is

TA= (B) + disp 0 ≤ disp ≤ 4095

Immediate Addressing Mode

- In this mode no memory reference is involved. If immediate mode is used the target address is the operand itself.

Indirect and PC-relative mode:

- In this type of instruction the symbol used in the instruction is the address of the location which contains the address of the operand. The address of this is found using PC-relative addressing mode.

(iii)Program Relocation:

- The assembler can identify for the loader those part of the object program that needed modification.
- An object program that contains the information necessary to perform this kind of modification is called **Relocatable program**.
- The instructions which are in the relative addressing modes (PC-relative or Base-relative) does not require any modification, wherever the program is loaded in the memory.

More than one program at a time sharing the memory and resources.

- Absolute program, starting address 1000
  - **e.g. 55 101B LDA THREE 00102D**
- Relocate the program to 2000
  - **e.g. 55 101B LDA THREE 00202D**
- Each Absolute address should be modified
- Modification record for each address field that needs to be changed when the program is relocated.

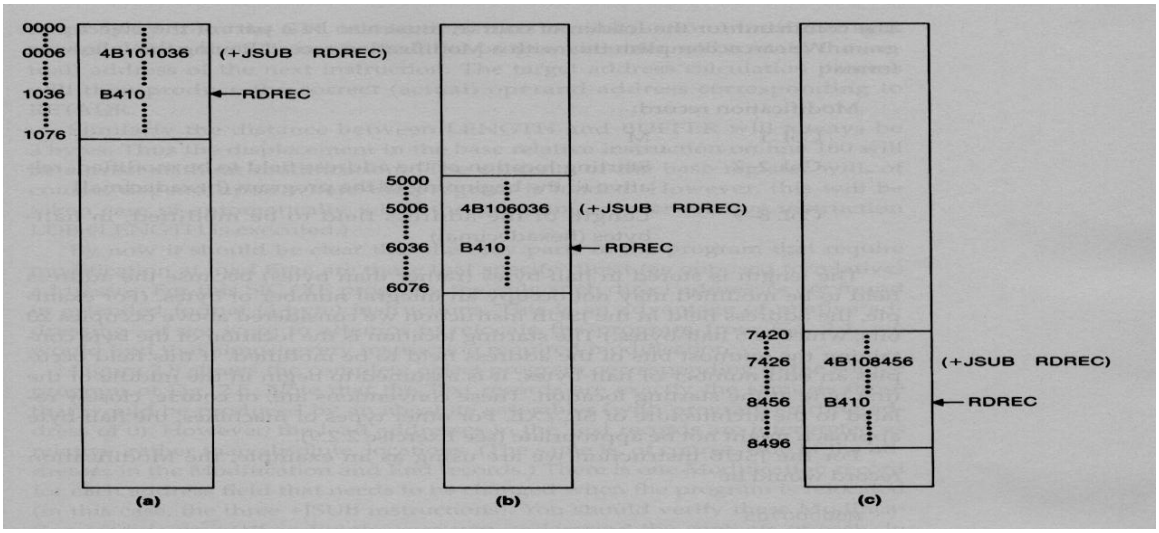
The format of Modification Record is

MODIFICATION RECORD:

Col 1	M
Col 2-7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal).
Col 8-9	Length of the address field to be modified in half bytes (hexadecimal).

- One modification record for each address to be modified.
- The length is stored in half-bytes (20 bits = 5 half-bytes).
- The starting location is the location of the byte containing the leftmost bits of the address field to be modified.
- If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

Example of Program Relocation



For example, a program is loaded beginning at address 0000. The JSUB instruction is loaded at address 0006. The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC.

11. Explain briefly about machine independent features of assembler? (11 marks)(NOV 2011, 2012)

These are the features which do not depend on the architecture of the machine. These are:

1. Literals
2. Symbol Defining Statement
3. Expressions
4. Program Blocks
5. Control Sections and Program Linking

(i)LITERALS:

- Literal is a constant operand which is used to write the value of it as a part of the instruction.
- This avoids having to define the constant elsewhere in the program and make up a label for it.
- A literal is defined with a prefix = followed by a specification of the literal value.

Example:

001A	ENDFIL	LDA	**=C'EOF'**	032010
		LTORG		
002D	*	**=C'EOF'**		454F46
1062	WLOOP	TD	**=X'05'**	E32011

Difference between Literal &Immediate Addressing

- With immediate addressing, the operand value is assembled as part of the instruction itself.

  Eg: LDA #3 010003

- With literal, the assembler generates specified value as a constant at some other memory location the address of this generated constant is used as target address for the machine instruction.

  Eg: ENDFIL LDA =C'EOF' 032010

Literal Pool:

- All the operands used in the program are gathered together into one or more literal pools.
- Normally literals are placed into a pool at the end of the program.
- In some cases it is desirable to place literals into a pool at some other location in the object program using the directive LTORG (LITERAL ORIGIN).
- When the assembler encounters the LTORG statement, it creates a literal pool that contains all of the literals operands used since the previous LTORG or the beginning of the program. Literals placed by LTORG will not be repeated at the pool at the end of the program .The duplicated literals are recognized by comparing the generated value of the constant operands.

Eg: usage of literals LDA = C'EOF'

LDA = X'454F46'

		Object code
Definition of literals	*= C'EOF'	454F46
	*=C'454F46'	454F46

- Literals can be used to refer to the current value of the location counter (address of the memory location).

  Eg: base *

  LDB =*
- The above instructions are used to specify the operand with value as current value of location counter.

(ii)SYMBOL DEFINING STATEMENT:

EQU Statement:

- Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.
- The directive used for this **EQU** (Equate).

The general form of the statement is

Symbol EQU value

- This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified.
- The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values.

For example

+LDT #4096

This loads the register T with immediate value 4096; this does not clearly what exactly this value indicates. If a statement is included as:

MAXLEN EQU 4096 and then

+LDT #MAXLEN

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table.

During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction.

Another common usage of EQU statement is for defining values for the general-purpose registers. The assembler can use the mnemonics for register usage like a-register A, X – index register and so on. But there are some instructions which require numbers in place of names in the instructions. For example in the instruction RMO 0, 1 instead of RMO A, X.

The programmer can assign the numerical values to these registers using EQU directive

```
A       EQU     0
X       EQU     1
```

These statements will cause the symbols A, X, L… to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed. As another usage if in a machine that has many general purpose registers named as R1, R2,…, some may be used as base register, some may be used as accumulator. Their usage may change from one program to another.

In this case we can define these requirement using EQU statements.

```
BASE    EQU     R1
INDEX   EQU     R2
COUNT   EQU     R3
```

One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined. For example, the following statement is not valid:

```
BETA EQU ALPHA ALPHA RESW 1
```

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

ORG Statement:

- This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin).
- Its general format is:

```
ORG value
```

Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG. ORG can be useful in label definition.

Suppose we need to define a symbol table with the following structure:

```
SYMBOL      6 Bytes
VALUE       3 Bytes
FLAG        2 Bytes
```

For example, if the symbol table is defined in the following structure:

STAB (100 entries)

SYMBOL	VALUE	FLAGS

In this table, the SYMBOL field contains a 6 byte user defined symbol. VALUE is a one word representation of the value assigned to the symbol. FLAGS is a 2 byte field that specifies symbol type and other information.

(iii) EXPRESSIONS:

- Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address.
- Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /.Division is usually defined to produce an integer

result. Individual terms may be

  - constants,
  - user-defined symbols, or
  - Special terms.

- The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location.

Thus the statement

**BUFFEND EQU *

Hence, expressions are classified as either

1. Absolute expression or
2. Relative expressions

Absolute Expression:

- The expression that uses only absolute terms is absolute expression.
- Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair.

Example:

MAXLEN EQU BUFEND-BUFFER

- The expression can have only absolute terms.

Example:

MAXLEN EQU 1000

- But produce the absolute value 1000 which is the length of buffer

Relative Expression:

- The value of relative expression is dependent of starting address of the program.
- It may contain relative terms or absolute terms but it must produce the value which is relative to the starting address of the program.

Example:

SYMBOL	**EQU**	**STAB**	
VALUE	**EQU**	**STAB+6** →	**1000+6=1006**
FLAGS	**EQU**	**STAB+9** →	**1000+9=1009**

- The value of the expression is 1006 which is some memory location within the program and also relative to the beginning address of the program.

(iv) PROGRAM BLOCKS:

- The program blocks refer to segments of code that are rearranged within a single object program unit.
- The control sections to refer to segments that are translated independent object program units.
- This feature allows more flexible handling of source and object programs.
- Here the generated machine instructions and data to appear in the object program in different order from the corresponding source statements.
- This results the creation of several independent parts of the object program.
- These parts maintain their identity and are handled separately by the loader.

Use Directives:

- USE Directive indicates which portion of the source program belongs to the various blocks.
- If no USE statements are included, the entire program belongs to the single block.
- This directive also indicate a continuation of previously begins block.
- Each program block contains several separate segments of the source program.
- The assembler with rearrange these segments to gather together the pieces of each block.
- The assembler accomplishes this logical rearrangement of code by maintaining a separate location counter for each program block during pass1.

Implementation of Program Blocks:

- The location counter for a block is initialized to 0 when the block is first begun.
- The current value of this location counter is saved when switching to another block and saved value is restored when resuming a previous block.

During Pass 1

- Each program block has a separate location counter.
- Each label is assigned an address that is relative to the start of the block that contains it.
- At the end of Pass 1, the latest value of the location counter for each block indicates the length of that block.
- The assembler can then assign to each block a starting address in the object program.

During Pass 2

- The address of each symbol can be computed by adding the assigned block starting address and the relative address of the symbol to that block.

For example, a program can be divided into 3 blocks namely:

1. **Default block** → This block contains executable instruction s of source program.
2. **CDATA** → This block includes storage area for few bytes.
3. **CBLCKS** → This block contains storage area for larger number of bytes.

- At the end of PASS1, the assembler constructs a table that contains the starting address and lengths of all blocks.

For example, it might be:

Block Name	Block No	Starting Address	Length
Default	0	0000	0066
CDATA	1	0066	000B
CBLCKS	2	0071	1000

- When the labels are entered in to the symbol table, the block name or number is stored along with its relative address.

Example symbol table:

Block no.	Symbol	value
0	FIRST	0000
0	CLOOP	0003
0	ENDFIL	0015
1	RETADR	0000
1	LENGTH	0003
2	BUFFER	0000
2	BUFEND	1000

(v) CONTROL SECTIONS AND PROGRAM LINKING:

- A **control section** is a part of the program that maintains its identity after assembly; each such control section can be loaded and relocated independently of others.
- Different control sections are most often used for subroutines or other logical subdivisions of a program.
- The programmer can assemble, load, and manipulate each of these control sections separately.
- The resulting flexibility is a major benefit of using control sections.
- When control sections form logically related parts of a program, it is necessary to provide some means for **linking** them together.
- For example, instructions in one control section might need to refer to instructions or data located in another section. Because control sections are independently loaded and relocated; the assembler is

  unable to process these references in usual way.
- The assembler has no idea where any other control section will be located at execution time. Such references between control sections are called **external references.**
- The assembler generated information for each external reference that will allow the loader to perform the required linking.

Assembler uses two assembler directives namely

1. **EXTDEF (EXTERNAL DEFINITION)**
   - The EXTDEF statement in a control section names symbols called external symbols that are defined in this control section and may be used by other sections.
2. **EXTREF (EXTERNAL REFERENCE)**
   - The EXTREF statement names symbols that are used in this control section and are defined elsewhere.
   - The assembler must also include the information about the external references in the object program that will cause the loader to calculate proper address of the operand and insert when

     they are required.
   - To include the information about the external references we need two record types in the object program.

The two new record types are

1. **Define record**
   - It gives information about external symbol that are defined in this control section that is symbol named by EXTDEF.
2. **Refer record**
   - It lists symbols that are used as external references by this control section but the symbols are defined in other control section, that is, symbol named by EXTREF.

FORMAT OF DERFINE RECORD:

Col 1	D
Col 2-7	name of external symbol defined in this control section
Col 8-13	relative address of symbol within this control section
Col 14-73	repeat information in col 2-13 for other external symbol

FORMAT OF REFER RECORD:

Col 1	R
Col 2-7	name of external symbol defined in this control section
Col 8-73	name of other external reference symbol

The new format of modification record is of the following

FORMAT OF MODIFICATION RECORD:

Col 1	M
Col 2-7	starting address of the field to be modified, relative to the beginning of the control section
Col 8-9	length of the field to be modified in half bytes
Col10	modification flag (+ or -)
Col 11-16	external symbol whose value is to be added to or subtracted from the indicated field

Example Object program:

```
H^COPY^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC^WRREC
T^000000610^172027^4B100000^032023^290000^32007^4B1000^3F2FFC^032016^0F2016
T^000010^00^010003^0F2000A^4B1000^3F2000
T^000030^03^454F46
M^000004^05+RDREC
M^000011^05+WRREC
M^000024^05+WRREC
E^000000
```

12. Explain Assembler design options? (11 marks) (MAY, NOV 2012)

The Assembler design options are

1. One –Pass Assemblers
2. Multi-Pass Assemblers

(i) ONE PASS ASSEMBLERS:

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

1. One that produces object code directly in memory for immediate execution **(Load- and-go assemblers).**
2. The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler

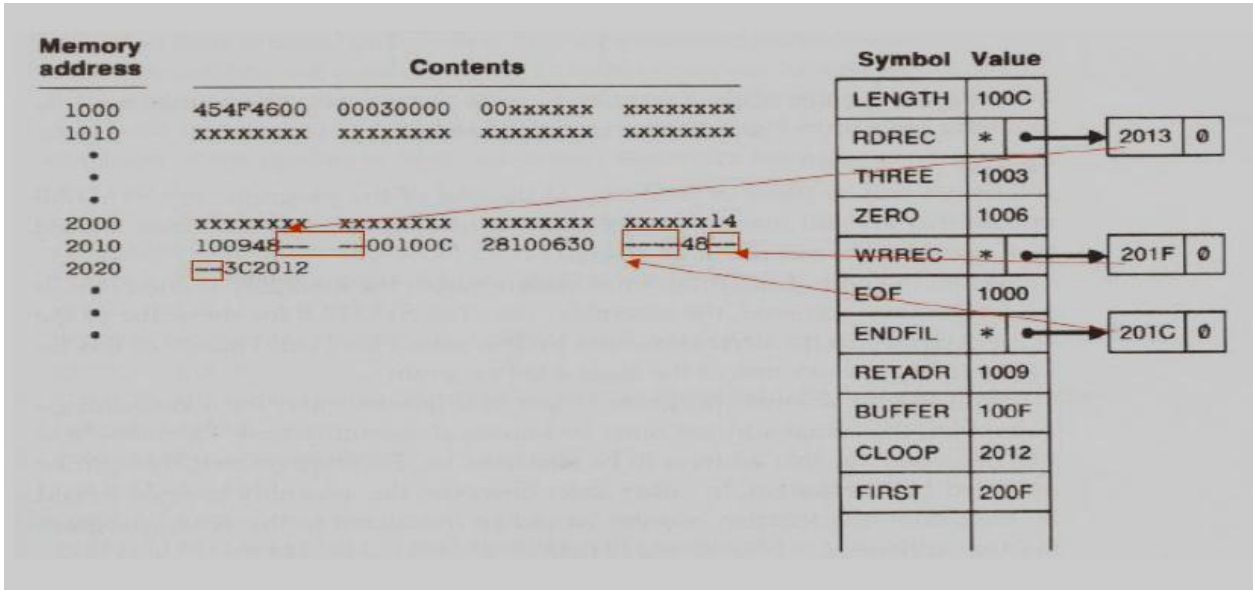
- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing.
- The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration**.**

Forward Reference in One-Pass Assemblers:

In load-and-Go assemblers when a forward reference is encountered:

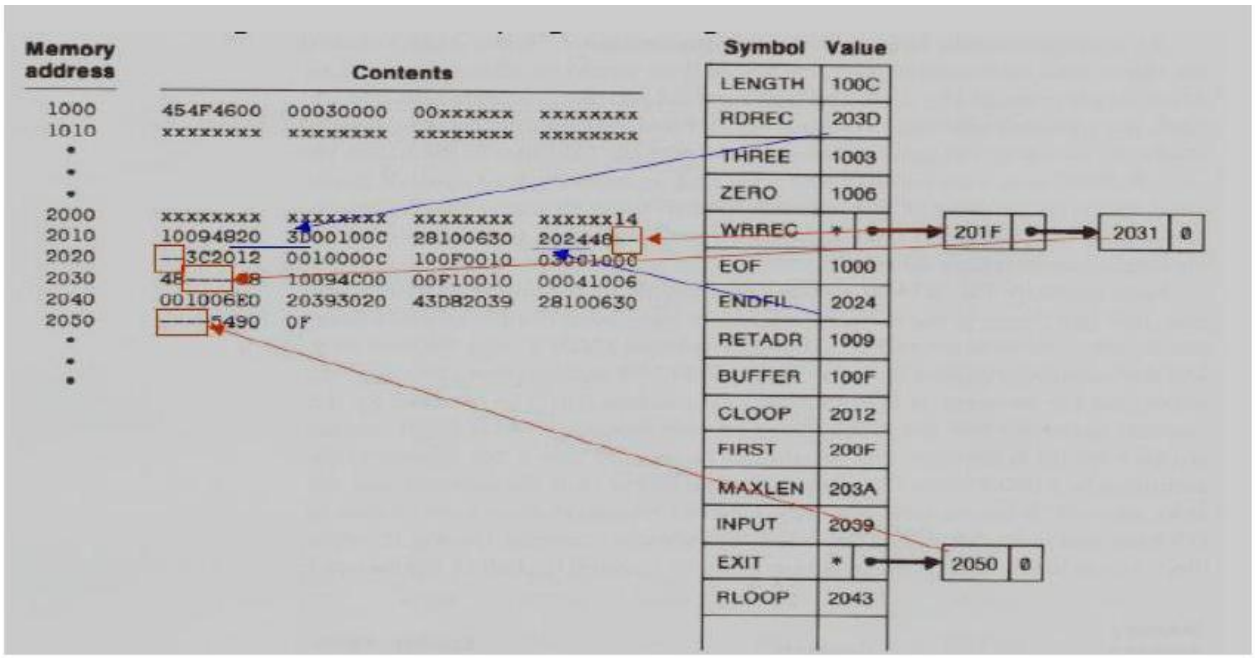
- Omits the operand address if the symbol has not yet been defined.
- Enters this undefined symbol into SYMTAB and indicates that it is undefined.
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry.
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.

- For Load-and-Go assembler
  - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error.



One-Pass needs to generate object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.



Object Code Generated by One-Pass Assembler:

```
HCOPY  001000001 07A
T001000094 54F46000003000000
T00200F15141009480000001 00C281006300000480000 3C2012
T00201C022024
T002024190010000C100F0010030C100C480000081009 4C0000F1001000
T00201302203D
T00203D1E041006001006E02039302043D82039281006300000 54900F2C203A382043
T002050022 05B
T00205B0710100C4C000005
T00201F022062
T002031022062
T00206218041006E02061302065 50900FDC20612C100C3820654C0000
E00200F
```

(ii) MULTI-PASS ASSEMBLERS:

- EQU assembler directive is required to define symbol used on the right hand side be defined in the source program.

For a two pass assembler, forward references in symbol definition are not allowed:

ALPHA	**EQU**	**BETA**
BETA	**EQU**	**DELTA**
DELTA	**RESW**	**1**

- Symbol definition must be completed in pass 1.

Prohibiting forward references in symbol definition is not a serious inconvenience.

- Forward references tend to create difficulty for a person reading the program.

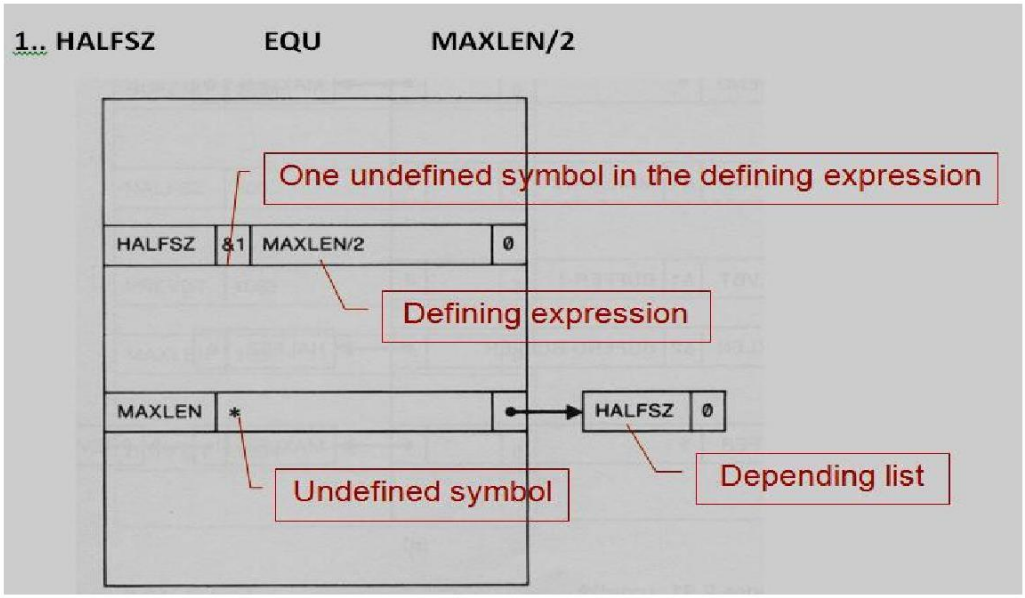
Implementation Issues for Modified Two-Pass Assembler:

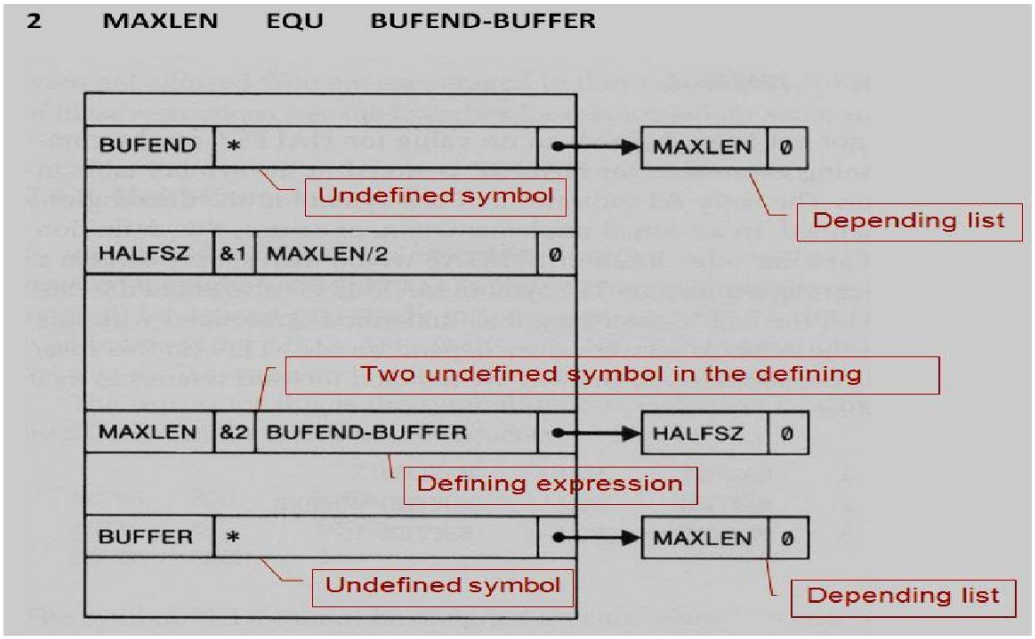
Implementation Issues when forward referencing is encountered in Symbol Defining statements:

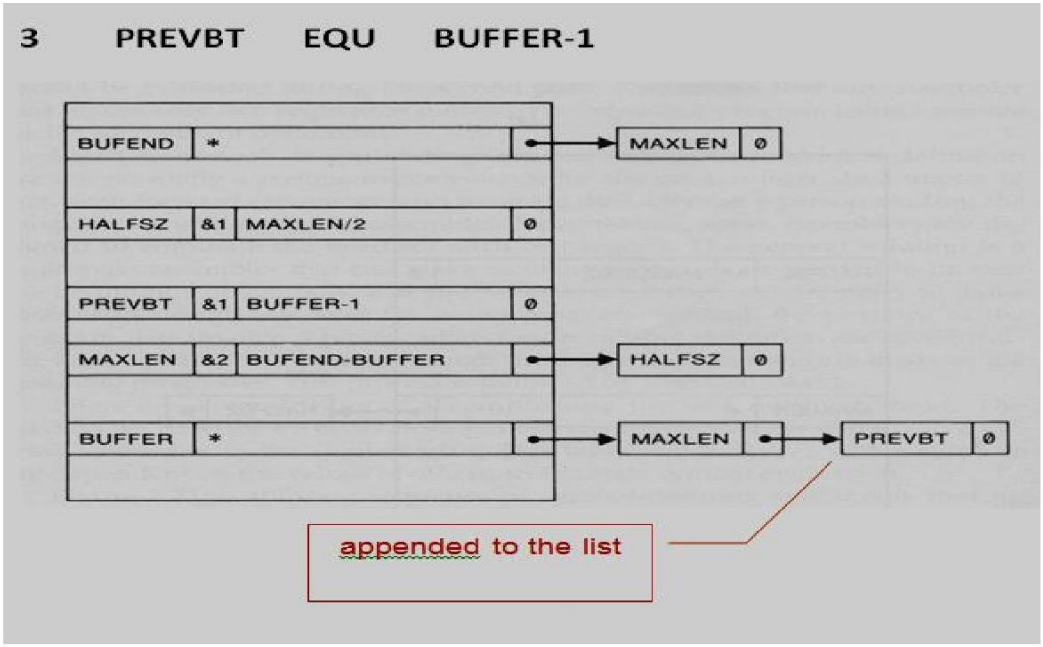
- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

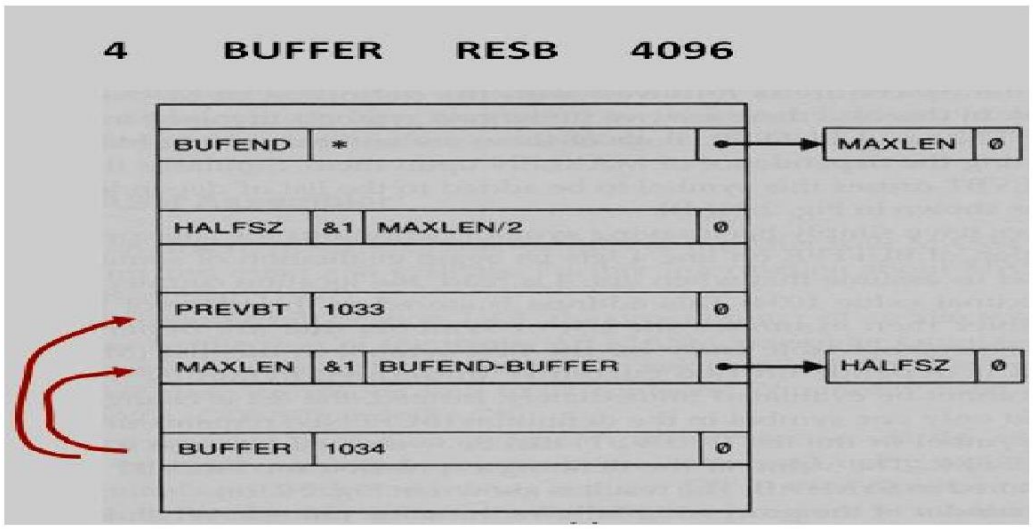
Multi-Pass Assembler Example Program:

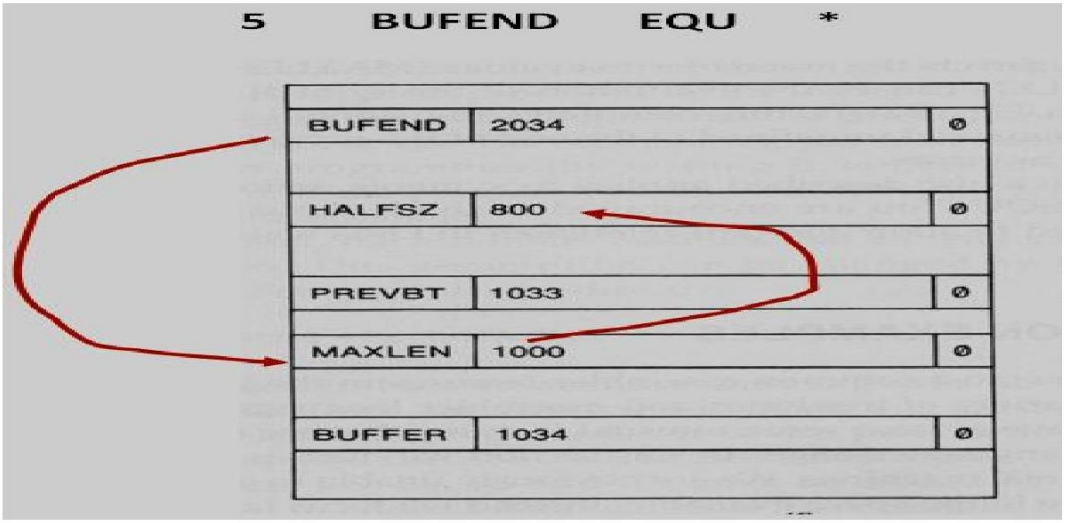
```
1       HALFSZ      EQU     MAXLEN/2
2       MAXLEN      EQU     BUFEND-BUFFER
3       PREVBT      EQU     BUFFER-1
                            .
                            .
                            .
4       BUFFER      RESB    4096
5       BUFEND      EQU     *
```






3 PREVBT EQU BUFFER-1
BUFEND * MAXLEN 0
HALFSZ &1 MAXLEN/2 0
PREVBT &1 BUFFER-1 0
MAXLEN &2 BUFEND-BUFFER HALFSZ 0
BUFFER * MAXLEN PREVBT 0
appended to the list


4 BUFFER RESB 4096
BUFEND * MAXLEN 0
HALFSZ &1 MAXLEN/2 0
PREVBT 1033 0
MAXLEN &1 BUFEND-BUFFER HALFSZ 0
BUFFER 1034 0


5 BUFEND EQU *
BUFEND 2034 0
HALFSZ 800 0
PREVBT 1033 0
MAXLEN 1000 0
BUFFER 1034 0

PONDICHERRY UNIVERSITY QUESTIONS

2 MARKS

1. What is interpreter?**(NOV 2011) (NOV 2012) (Ref.Qn.No.10, Pg.no.6)**
2. What are the fields available in an assembly language instruction? **(NOV 2011) (Ref.Qn.No.28, Pg.no.10)**
3. Define Assembler?**(MAY 2012) (Ref.Qn.No.9, Pg.no.5)**
4. Define System Software. ?**(MAY 2012) (NOV 2012) (Ref.Qn.No.4, Pg.no.4)**
5. What is mean by OPTAB, SYMTAB? **(MAY 2012) (Ref.Qn.No.33, Pg.no.11)**
6. List the assembler directives. **(NOV 2012) (Ref.Qn.No.29, Pg.no.10)**
7. Define Instruction set. **(MAY 2013) (Ref.Qn.No.17, Pg.no.7)**
8. How could literals be implemented in one pass assembler? **(MAY 2013) (Ref.Qn.No.12, Pg.no.6)**
9. State the difference between assembler and Interpreter. **(NOV 2013) (Ref.Qn.No.11, Pg.no.6)**
10. List the features of machine dependent Assembler. **(NOV 2013) (Ref.Qn.No.36, Pg.no.12)**

11 MARKS

NOV 2011(REGULAR)

1. (a) Draw the format of an instruction and explain (5) **(Ref.Qn.No.5,6 Pg.no.24,26)**

   (b) Describe the machine structure. (6) **(Ref.Qn.No.5, Pg.no.24)**

   (OR)

2. Describe the features of machine-independent assembler. **(Ref.Qn.No.11, Pg.no.43)**

MAY 2012(ARREAR)

1. List out and discuss the machine dependent assembler features. **(Ref.Qn.No.10, Pg.no.39)**

   (OR)

2. Briefly explain about assembler design options. **(Ref.Qn.No.12, Pg.no.51)**

NOV 2012(REGULAR)

1. List out and discuss the machine independent assembler features. **(Ref.Qn.No.11, Pg.no.43)**

   (OR)

2. Briefly explain about one-pass assembler and multi-pass assembler. **(Ref.Qn.No.12, Pg.no.51)**

MAY 2013(ARREAR)

1. What is the important machine structures used in the design of system software? Discuss.

   (Ref.Qn.No.5, Pg.no.24) **(OR)**

2. Give two example programs SIC operations. **(Ref.Qn.No.7, Pg.no.29)**

NOV 2013 (REGULAR)

1. Explain the simplified instructional computer (SIC) architecture in detail. **(Ref.Qn.No.5, Pg.no.24)**

   (OR)

2. Explain the basic assembler functions and the steps in the design of an assembler. **(Ref.Qn.No.8, Pg.no.31)**

SRI VENKATESHWARAA COLLEGE OF ENGINEERING & TECHNOLOGY
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University, Puducherry.)
13-A, Villupuram – Pondy Main road, Ariyur, Puducherry – 605 102.
Phone: 0413-2644426, Fax: 2644424 / Website: www.svcetpondy.com

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: LANGUAGE TRANSLATORS SUBJECT CODE: CST52

UNIT II

Loaders and Linkers: Basic loader functions, machine–dependent and machine–independentloader features. Loader design – Linkage editors, dynamic linking and bootstrap loaders.

2 MARKS

1. Define loader.

- A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. Many loaders also support relocation and linking.
- Loader is a set of program that loads the machine language translated by the translator into the main memory and makes it ready for execution.

2. Define linker or linkage editor.

- Linker or linkage editor to perform the linking operations and a separate loader to handle relocation and loading.

3. What are the two primary tasks of a linker? (NOV 2013)

The two primary tasks of the linker are

1. Relocating relative addresses
2. Resolving external references

4. What are the basic functions of loaders?

- Loading – brings the object program into memory for execution.
- Relocation – modifies the object program so that it can be loaded at an address different from the location originally specified.
- Linking – combines two or more separate object programs and supplies the information needed to references between them.

5. What are the types of loaders?

The different types of loaders are

1. absolute loader
2. bootstrap loader
3. relocating loader or relative loader
4. linking loader.

6. What is an absolute loader? State its disadvantages. (MAY 2013)

- The loader, which is used only for loading, is known as absolute loader.
- The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program.
- e.g. Bootstrap loader

Disadvantages:

The disadvantages are,

1. It is need for programmer to specify the actual address at which it will be loaded into memory.
2. It is difficult to use subroutine libraries efficiently.

7. What is the design of absolute loader?

This loader does not perform such functions as linking and program relocation. Its operation is very similar to all functions are accomplished in a single pass.

- The Header record is checked to verify that the correct program has been presented for loading.
- Each Text record is read, the object code contains is moved to the indicated address in memory.
- When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

8. Write the algorithm for absolute loader.

```
begin
    read Header record
    verify program name and
    length read first text record
    while record type ≠ 'E'
        do begin
            { if object code is in character form, convert
            into internal representation }
            move object code to specified location in
            memory read next object program record
        end
    jump to address specified in End record
end
```

9. What is bootstrap loader? (MAY 2012)

- When a computer is first tuned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer – usually an

  operating system.
- It then jumps to the just loaded program to execute it .The bootstrap itself begins at address 0.
- It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

10. What is program relocation?

- Program relocation is the method of loading the object code in a different location specified in the object program. Assembler passes this information about the program relocation to the loader through modification record as a part of object program.
- Program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time.

11. What is address sensitive program? (NOV 2011)

- The direct addressing mode is called address sensitive program.
- When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required.
- Direct addressing ***E.g. LDA ZERO***

12. What are relocating loaders or relative loaders?

Loaders that allow for program relocation are called relocating loaders or relative loaders.

13. List out the functions performed by relocating loaders? (NOV 2011)

The two functions performed by relocation loaders are

1. Modification record and
2. Relocation bit.

- In modification record, a modification record M is used in the object program to specify any relocation.
- In relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

14. What is the use of modification record?

Modification record is used for program relocation. Each modification record specifies the starting address and the length of the field whose value is to be altered and also describes the modification to be performed.

15. Define Relocation bit.

- Relocation bit associated with each word of object code. In SIC instructions occupy one word; this means that there is one relocation bit for each possible instruction.
- The relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the program is relocated.
- Bit value 0 indicates no modification is required.

16. Define bit mask.

- The relocation bits are gathered together following the length indicator in each text record and which is called as bit mask.
- For e.g. the bit mask FFC (111111111100) specifies that the first 10 words of object code are to be modified during relocation.

17. What is the need of ESTAB?

- External Symbol Table (ESTAB) is used to store the name and address of the each external symbol.
- It also indicates in which control section the symbol is defined.

18. What is the use of PROGADDR?

- Program load Address (PROGADDR) is the beginning address in memory where the linked program is to be loaded.
- Its value is supplied to the loader by the operating system.

19. What is the use of CSADDR?

- Control Section Address (CSADDR) contains the starting address assigned to the control section currently being scanned by the loader.
- Its value is added to all relative addresses within the control section to convert them to actual addresses.

20. Write the two passes of a linking loader.

The two passes of linking loaders are

- Pass1: assigns address to all external symbols.
- Pass2: it performs the actual loading, relocation and linking.

21. How are duplicate literal operands handled in machine independent loader? (NOV 2013)

- The duplicate literal used more than once in the program. Only one copy of the specified value needs to be stored.

For how to recognize the duplicate literals

- Compare the character strings defining them. Easier to implement, but has potential problem
- ***E.g., =X'05'***

22. Define automatic library search.

In many linking loaders the subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. This feature is referred to as automatic library search. Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

23. List the loader options INCLUDE &DELETE.

The general format of INCLUDE is

INCLUDE program_name (library name)

- This command directs the loader to read the designated object program from a library and treat it as the primary loader input.

The general format of DELETE command is

DELETE csect-name

- It instructs the loader to delete the named control sections from the sets of programs loaded.

24. What are the primary loader commands?

The primary loader input with the loader commands are

- INCLUDE READ (UTLIB)
- INCLUDE WRITE (UTLIB)
- DELETE RDREC, WRREC
- CHANGE RDREC, READ
- CHANGE WRREC, WRITE

25. List the loader design options. 1. Linkage Editors

2. Dynamic linking

3. Bootstrap loaders

26. Give the functions of the linking loader.

The linking loader performs all linking and relocation operations, including automatic library search and loads the linked programs directly loaded into the memory for execution.

27. Define link editor? (MAY 2013)

- A linkage editor produces a linked version of the program (often called a **load module** or an **executable image**), which is written to a file or library for later execution.
- It performs relocation of all control sections relative to the start of the linked program.

28. Why is linking required after a program is translated? (MAY 2013)

- Linking which combines two or more separate object programs and supplies the information needed to allow references between them
- The routines are automatically retrieved from a library as they are needed during linking.

29. The difference between Linkage Editor and Linking Loader?

Linking Loader	Linkage Editor
A linking loader performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution.	A linkage editor produces a linked version of the program, which is normally written to a file for later execution.
Linkage editors perform linking operations before the program is loaded for execution	Linking loaders perform these same operations at load time.

30. Define dynamic linking.

- Dynamic linking postpones the linking function until execution time; a subroutine is loaded and is linked to the rest of the program when it is first called(run time).This type of function is usually called

  dynamic loading or **dynamic linking** or **load on call.**

31. Write the advantage of dynamic linking.

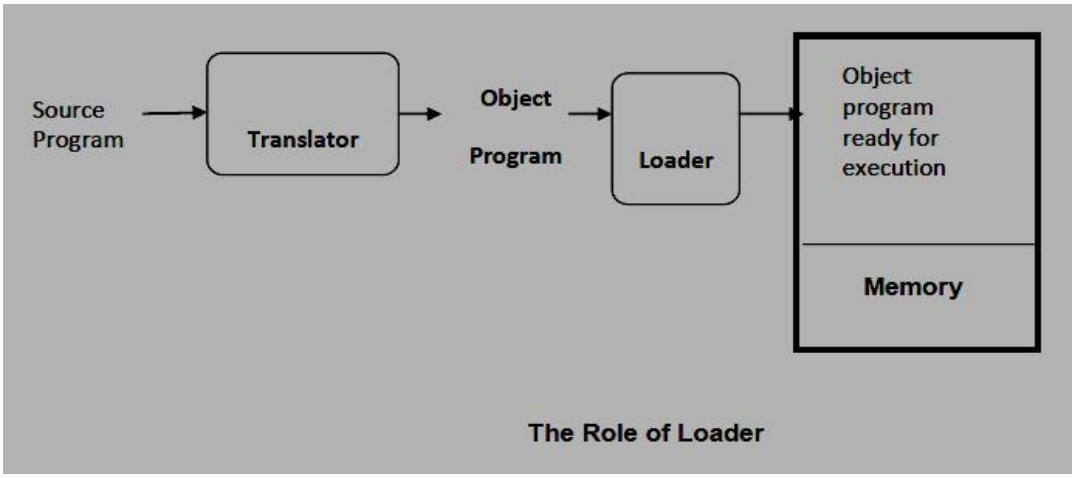
- Dynamic linking is used to allow several executing programs to share one copy of a subroutine or library.
- It is often used for references to software object.
- It has the ability to load the routine only when they are needed.
- The dynamic linking avoids the loading of entire library for each execution.

11 MARKS

1. Explain with neat block diagram the role of loader and linker. (11 marks)(MAY 2013)

Role of Loader

- A *loader* is a system program that performs the loading function. It brings object program into memory and starts its execution. Many loaders also support relocation and linking.
- Loader is a set of program that loads the machine language translated by the translator into the main memory and makes it ready for execution.
- An object program contains translated instructions and data values from the source program and specifies addresses in memory where these items to be loaded.
- The loader is a program which accepts the object program checks, prepares these programs for execution by the computer and initiates the execution.



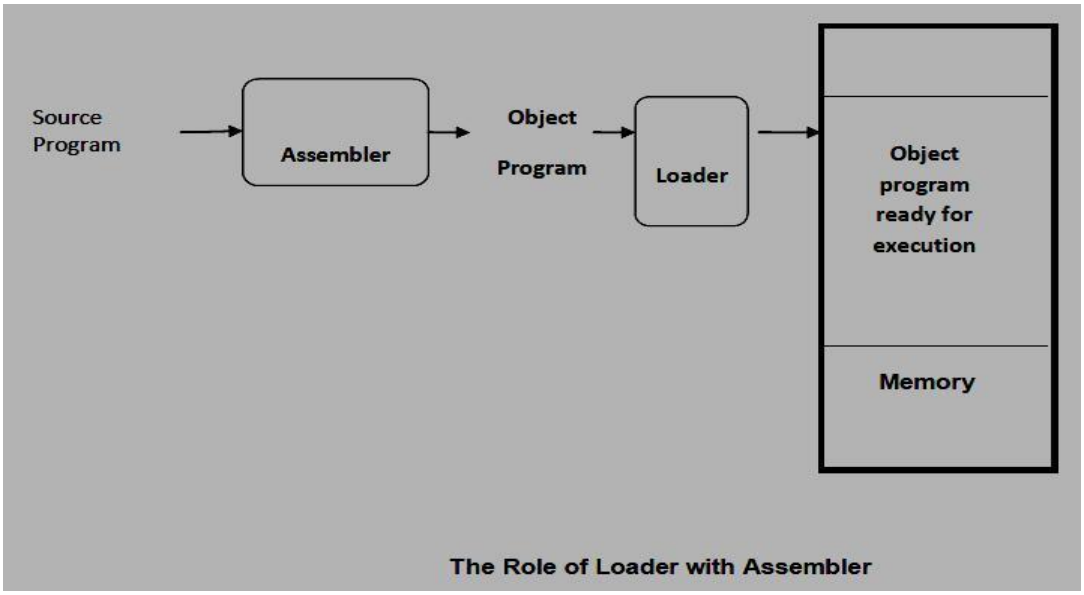
The Role of Loader

The loader must perform the following functions:

1. *Loading*: This brings the object program into memory for execution.
2. *Relocation:* This modifies the object program so that it can be loaded at an address from thelocation originally specified.
3. *Linking:* This combines two or more separate object programs and supplies the informationneeded to allow references between them.

- A loader is a system program that performs the loading function. Many loaders also support relocation and linking. Some systems have a *linker or linkage editor* to perform the linking and a separate loader to handle relocation and loading.
- In most cases all the program translators (*assemblers and compilers*) on a particular system produce object programs.

Role of loader with assembler:



The Role of Loader with Assembler

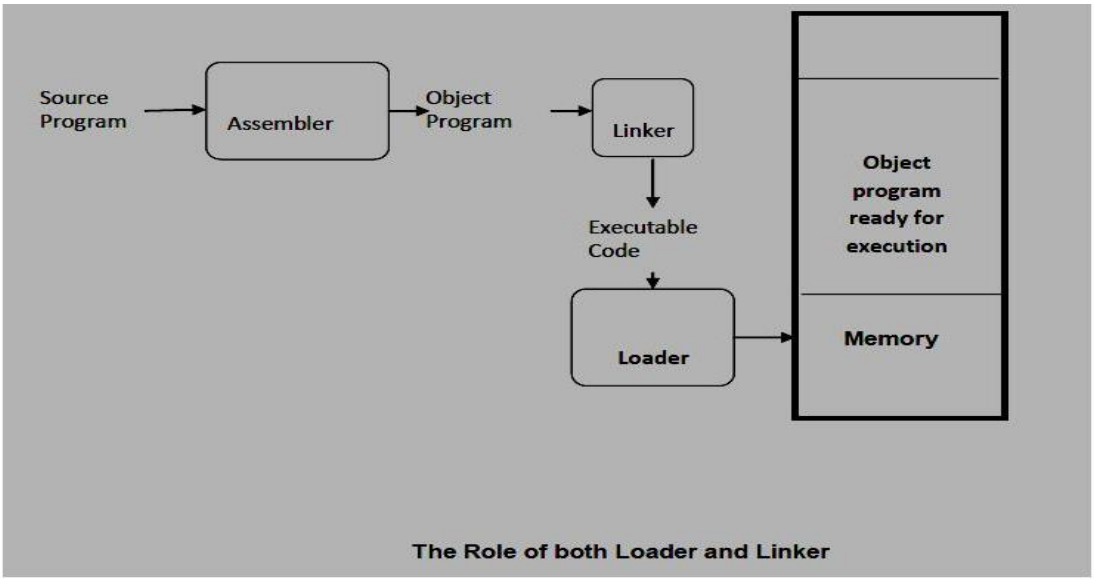
Type of Loaders

The different types of loaders are,

1. absolute loader
2. bootstrap loader
3. relocating loader or relative loader
4. linking - loader

Role of loader and linker:

- Linker combines two or more separate object programs and supplies the information needed to allow references between them
- Linker or linkage editor to perform the linking operations and a separate loader to handle relocation and loading.



The Role of both Loader and Linker

2. Discuss about Basic Loader Functions. (MAY 2012) or Discuss the functions and design of an absolute loader. (NOV 2011)

Basic loader functions:

- The most fundamental function of the loader is bringing the object program into memory and starts its execution.
    1. Design of an absolute Loader
    2. A simple Bootstrap Loader

Design of an Absolute Loader:

This loader does not perform such functions as linking and program relocation, its operation is very simple. All functions are accomplished in a single pass.

- The Header record is checked to verify that the correct program has been presented for loading.
- As each Text record is read, the object code contains is moved to the indicated address in memory.
- When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

Algorithm for an Absolute Loader:

```
begin
    read Header record
    verify program name and
    length read first text record
    while record type ≠ 'E'
        do begin
            { if object code is in character form, convert
            into internal representation }
            move object code to specified location in
            memory read next object program record
        end
    jump to address specified in End record
end
```

Representation of object program:

```
HCOPY  001000001007A
T001000 1E 14103348 20390010 36281030 30101548 20613C10 0300102A 0C103900 102D0C10 36482061 0810334C 0000454F 46000003 000000xx
T00101E 150C1036 48206108 10334C00 00454F46 000003000000
T002039 1E041030 001030E0 205D3020 3FD8205D 28103030 20575490 392C205E 38203F10 10364C00 00F10010
T002057 1C101036 4C0000F1 001000041030E020 79302064 509039DC 20792C1036
T002073 07382064 4C000005
E001000
```

(a) Object program

The contents of memory locations for which there is no text record as shown as **xxxx**

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
1000	14103348	20390010	36281030	30101548	
1010	20613C10	0300102A	0C103900	102D0C10	
1020	36482061	0810334C	0000454F	46000003	
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	← COPY
⋮	⋮	⋮	⋮	⋮	
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0	
2040	205D3020	3FD8205D	28103030	20575490	
2050	392C205E	38203F10	10364C00	00F10010	
2060	00041030	E0207930	20645090	39DC2079	
2070	2C103638	20644C00	0005xxxx	xxxxxxxx	
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

(b) Program loaded in memory

- In the object program, each byte of assembled code is given using its hexadecimal representation in character form.
- The operation code for an **STL** instruction would be represented by the pair of characters **"1"** and **"4"**.
- When these are read by the loader, it will occupy two bytes of memory.
- In the instruction as loaded for execution, however, this operation code must be stored in a single byte with hexadecimal value 14.
- Thus each pair of bytes from the object program record must be packed together into one byte during loading.

Disadvantages of Absolute Loader

- The program must specify to the assembler the address in memory where the program is to be loaded.
- If there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in his other subroutines to perform subroutine linkage.
- The programmer must be careful not to assign two subroutines to the same or overlapping.

3. Define bootstrap loader and state the algorithm. (NOV 2011) (MAY 2012, 2013)

- When a computer is first tuned on or restarted, a special type of absolute loader, called *bootstraploader* is executed. This bootstrap loads the first program to be run by the computer– usually anoperating system.
- The bootstrap itself begins at address 0 in the memory of the machine.
- It loads the operating system starting at address 80.
- No header record or control information, the object code is consecutive bytes of memory.

Algorithm for bootstrap loader

The algorithm for the bootstrap loader is as follows

begin

X=0x80 (the address of the next memory location to be loaded

Loop

A GETC (and convert it from the ASCII character code to the value of the hexadecimal digit) save the value in the high-order 4 bits of S
A GETC
combine the value to form one byte A (A+S)
store the value (in A) to the address in register
X X X+1

End

It uses a subroutine GETC, which is

GETC A read one character

if A=0x04 then jump to

0x80 if A<48 then GETC

A A-48 (0x30)
if A<10 then
return A A-7
return

- Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits.
- However, there is no Header record, End record or control information (such as addresses or lengths).
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

4. Explain Machine Dependent Loader Features? (11 marks)

The design and implementation of a more complex loader. This loader provides for program relocation and linking. For efficient sharing of the machines, we go for relocatable loader instead of absolute ones. This relocatable loader provides for program relocation and linking.

It contains

i. Program Relocation

ii. Program linking

iii. Algorithms and data structures

(i) PROGRAM RELOCATION:

Loaders that allow for program relocation are called *relocating loaders or relative loaders*. There are two methods for specifying relocation as part of the object program.

a) Using Modification Record.

b) Using relocation bit mask.

a) Program relocation using modification record:

- A modification record is used to describe each part of the object code that must be changed when the program is relocated.
- Most of the instructions in this program use relative or immediate addressing.
- The only portions of the assembled program that contain actual addresses are the extended format instructions.
- There is one modification record for each value that must be changed during relocation.
- Each modification record specifies the starting address and length of the field whose value is to be altered.
- It then describes the modification to be performed.

Eg.**M^ 000007 ^ 05 + COPY.**

- In this example, modification adds the value of the symbol COPY which represents the starting address of the program.
- The modification record scheme is a convenient means for specifying the program relocation.

SIC/XE Relocation Loader

algorithm begin

get PROGADDR from operating system **while** not end of input **do**

begin

read next record

whilerecord type ≠ 'E'

do begin

read next input record

while record type= 'T'

then

begin

move object code from record to location ADDR + specified address. **end**

whilerecord type = 'M'

add PROGADDR at the location PROGADDR + specified address.

end

end

end

b) Program relocation using Bit Mask

- Relocation bit associated with each word of object code. In SIC instructions occupy one word; this means that there is one relocation bit for each possible instruction.
- The relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the program is relocated.
- Bit value 0 indicates no modification is required.
- The relocation bits are gathered together following the length indicator in each text record and which is called as *bit mask.*

```
HCOPY  000000001 07A
T0000001EFFC140033481039000036280030300015481061 3C000300002A0C003900002D
T00001E15E000C003648106108003 34C0000454F46000003000000
T001039 1EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

If a text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0. Thus the bit mask FFC (111111111100) in the first text record specifies that all 10 words of the object code are to be modified during relocation. The mask E00 in the second text record specifies that the first 3 words are to be modified.

SIC Relocation Loader Algorithm

```
begin
    get PROGADDR from operating
  system while not end of input do
    begin
        read next record while
      record type ≠ 'E' do
      while record type = 'T'
        begin
            get length = second data
            mask bits(M) as third
            data
          for( i=0 ; i< length;
              i++) if Mi= 1
              then
                  add PROGADDR at the location PROGADDR + specified address
              else
                  move object code from record to location PROGADDR + specified
                  address.
            read next record
        end
    end
 end
```

(ii) PROGRAM LINKING:

- A program may be composed of many control sections.
- These control sections may be assembled separately.
- These control sections may be loaded at different addresses in memory.
- External references to symbol defined in other control sections can only be resolved after these control sections are loaded into memory.

Example

- Let us consider three separately assembled programs, each of which consists of a single control section.
- Each program contains a list of items (LISTA, LISTB, LISTC).
- The ends of these lists are marked by the labels ENDA, ENDB, ENDC.

The labels on the beginnings and ends of the lists are external symbols. That is they are available for use in linking. Each program contains exactly the same set of references to these external symbols.

External Symbols

- ***EXTDEF (external definition)*** - The EXTDEF statement in a control section names symbols, calledexternal symbols, that are defined in this (present) control section and may be used by other sections. **ex:** EXTDEF LISTA, ENDA

  EXTDEF BUFFER, BUFFEND, LENGTH

- ***EXTREF (external reference)*** - The EXTREF statement names symbols used in this (present) controlsection and are defined elsewhere.

  ex: EXTREF LISTB, ENDB, LISTC,

  ENDCEXTREF RDREC,

  WRREC

Example in program linking

Program A

```
0000   PROGA     START     0
                  EXTDEF    LISTA, ENDA
                  EXTREF    LISTB, ENDB, LISTC, ENDC
                  ----------
                  ----------
0020   REF1       LDA       LISTA                    03201D
0023   REF2       +LDT      LISTB+4                  77100004
0027   REF3       LDX       #ENDA-LISTA              050014
                     .
                     .
0040   LISTA      EQU       *

0054   ENDA       EQU       *
0054   REF4       WORD      ENDA-LISTA+LISTC         000014
0057   REF5       WORD      ENDC-LISTC-10            FFFFF6
005A   REF6       WORD      ENDC-LISTC+LISTA-1       00003F
005D   REF7       WORD      ENDA-LISTA-(ENDB-LISTB)  000014
0060   REF8       WORD      LISTB-LISTA              FFFFC0
                  END       REF1
```

Program B

```
0000   PROGB     START     0
                  EXTDEF    LISTB, ENDB
                  EXTREF    LISTA, ENDA, LISTC, ENDC
                  ----------
                  ----------
0036   REF1       +LDA      LISTA                    03100000
003A   REF2       LDT       LISTB+4                  772027
003D   REF3       +LDX      #ENDA-LISTA              05100000
                     .
                     .
0060   LISTB      EQU       *

0070   ENDB       EQU       *
0070   REF4       WORD      ENDA-LISTA+LISTC         000000
0073   REF5       WORD      ENDC-LISTC-10            FFFFF6
0076   REF6       WORD      ENDC-LISTC+LISTA-1       FFFFFF
0079   REF7       WORD      ENDA-LISTA-(ENDB-LISTB)  FFFFF0
007C   REF8       WORD      LISTB-LISTA              000060
                  END
```

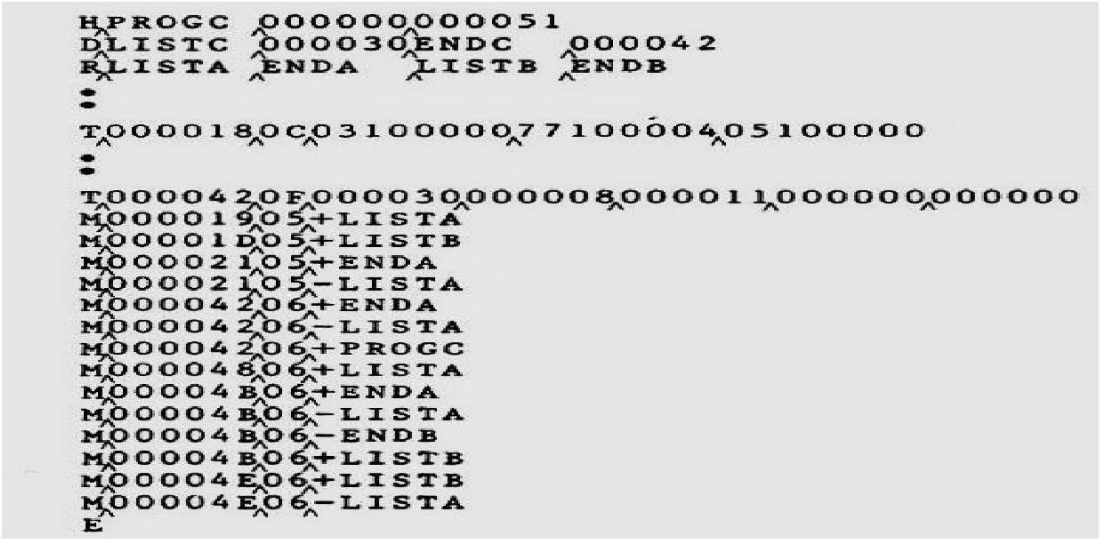
Program C

```
Loc       Source statement                              Object code

0000      PROGC   START   0
                  EXTDEF  LISTC,ENDC
                  EXTREF  LISTA,ENDA,LISTB,ENDB
                  .
                  .
                  .
0018      REF1    +LDA    LISTA                         03100000
001C      REF2    +LDT    LISTB+4                       77100004
0020      REF3    +LDX    #ENDA-LISTA                   05100000
                  .
                  .
                  .
0030      LISTC   EQU     *
                  .
                  .
0042      ENDC    EQU     *
0042      REF4    WORD    ENDA-LISTA+LISTC              000030
0045      REF5    WORD    ENDC-LISTC-10                 000008
0048      REF6    WORD    ENDC-LISTC+LISTA-1            000011
004B      REF7    WORD    ENDA-LISTA-(ENDB-LISTB)       000000
004E      REF8    WORD    LISTB-LISTA                   000000
                  END
```

Object Program Example

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
:
T0000200A03201D771000040050014
:
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
:
T0000360B03100000772027005100000
:
T0000700F000000FFFFF6FFFFFFFFFF0000060
M00003705+LISTA
M00003E05+ENDA
M00003E05-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007906+ENDA
M00007906-LISTA
M00007C06+PROGB
M00007C06-LISTA
E
```



- Notice that program A defines LISTA and ENDA, program B defines LISTB and ENDB, and program defines LISTC and ENDC.
- Notice that the definitions of REF1, REF2, .., to REF7 in all of these three control sections are the same.
- Therefore, after these three control sections are loaded, no matter where they are loaded, the values of REF1 to REF7 in all of these programs should be the same.

REF 1 LISTA

- Program A
  - LISTA is defined in its own program and its address is immediately available. Therefore, wecan simply use program counter-relative addressing
- Program B
  - Because LISTA is an external reference, its address is not available now. Therefore anextended-format instruction with address field set to 00000 is used. A modification record in inserted into the object code so that once LISTA's address is known, it can be added to this field.
- Program C
  - The same as that processed in Program B.

REF 2 LISTB + 4

- Program A
  - Because LISTB is an external reference, its address is not available now. Therefore anextended-format instruction with address field set to 00004 is used. A modification record is inserted into the object code so that once LISTB's address is available; it can be added to this field.

- Program B
  - LISTB is defined in its own program and its address is immediately available. Therefore, wecan simply use program counter-relative addressing
- Program C
  - The same as that processed in Program A.

REF 3 # ENDA - LISTA

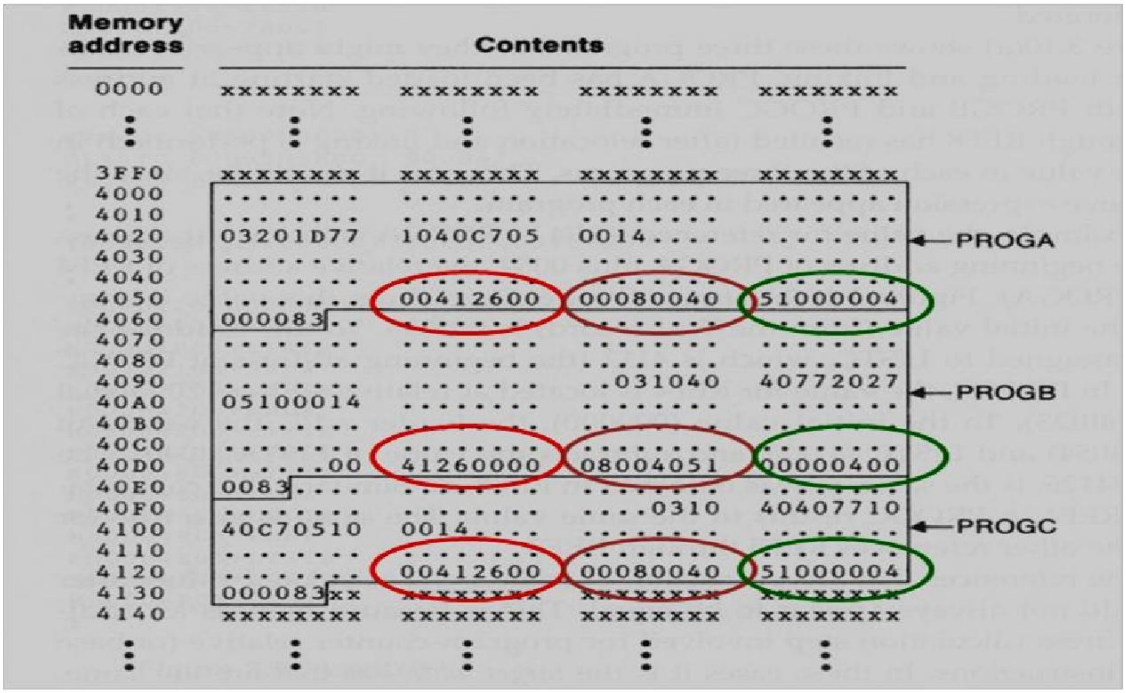
- Program A
  - The difference between ENDA and LISTA (14) is immediately available during assembly.
- Program B
  - Because the values of ENDA and LISTA are unknown during assembly, we need to use anextended-format instruction with its address field set to 0.
  - Two modification records are inserted to the object program–one for +ENDA and the otherfor –LISTA.
- Program C
  - The same as that processed in Program B.

REF4 ENDA – LISTA + LISTC

- Program A
  - The difference between ENDA and LISTA can be known now. Only the value of LISTC isunknown. Therefore, an initial value of 000014 is stored with one modification record for LISTC.
- Program B
  - Because none of ENDA, LISTA, and LISTC's values can be knownnow, an initial value of000000 is stored with three modification records for all of them.
- Program C
  - The value of LISTC is known now. However, the values for ENDA and LISTA are unknown. Aninitial value of 000030 is stored with two modification records for ENDA and LISTA.

After Loading into Memory

- Suppose that program A is loaded at 004000, program B at 004063, and program C at 0040E2.
- The REF4, REF5, REF6, and REF7 in all of these three programs have the same values.



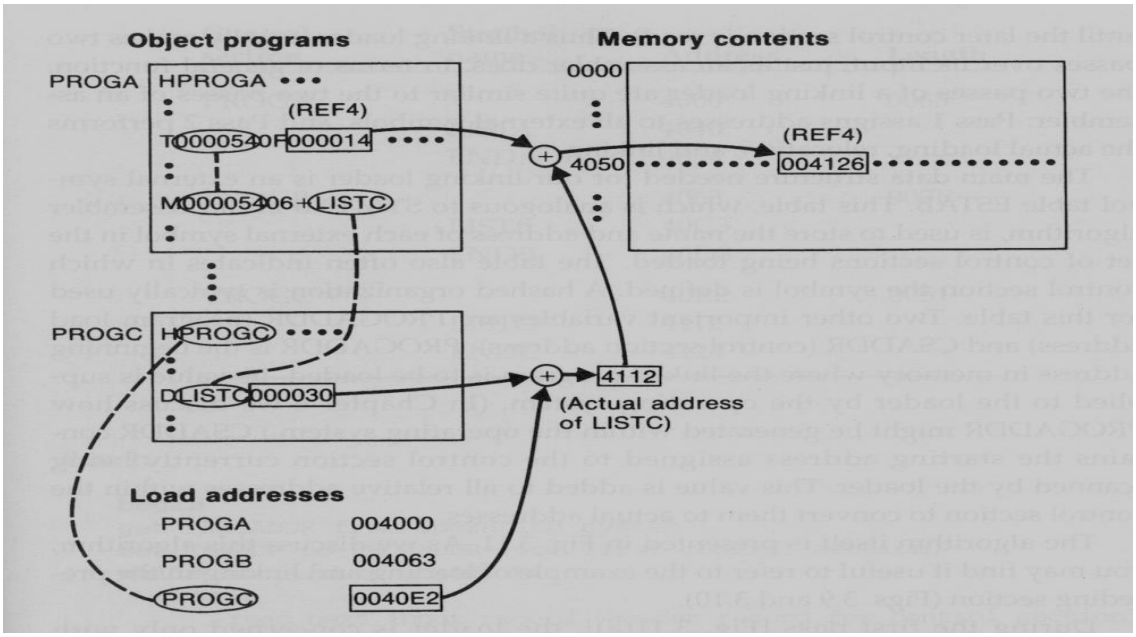
REF4 AFTER LINKING

In Program A

– The address of REF4 is 4054 (4000 + 54) because program A is loaded at 4000 and the relativeaddress of REF4 within program A is 54.

– The value of REF4 is 004126 because

- The address of LISTC is 0040E2 (the loaded address of program C) + 000030 (the relative address of LISTC in program C)
- 0040E2 + 000014 (constant already calculated) = 004126.



- **In Program B**
  - The address of REF4 is 40D3 (4063 + 70) because program B is loaded at 4063 and the relativeaddress of REF4 within program A is 70.
  - The value of REF4 is 004126 because
    - The address of LISTC is 004112
    - The address of ENDA is 004054
    - The address of LISTA is 004040
    - 004054 + 004112 – 004040 = 004126

(iii) ALGORITHM AND DATASTRUCTURES OF A LINKING LOADER:

Implementation of Loader and Linker

A linking loader makes two passes over its input

- In pass 1: assign addresses to external references
- In pass 2: perform the actually loading, relocation, and linking

Data Structures

- **External symbol table (ESTAB)**
  - Like SYMTAB, store the name and address of each external symbol in the set of control sectionsbeing loaded.
  - It needs to indicate in which control section the symbol is defined.
- **Program load Address (PROGADDR)**
  - The beginning address in memory where the linked program is to be loaded.
  - Its value is supplied to the loader by the operating system.
- **Control Section Address (CSADDR)**
  - It contains the starting address assigned to the control section currently being scanned by theloader.
  - This value is added to all relative addresses within the control sections.

Algorithm – PASS1

- During pass 1, the loaders are concerned only with HEADER and DEFINE record types in the control sections to build ESTAB.
- PROGADDR is obtained from OS.
- This becomes the starting address (CSADDR) for the first control section.
- The control section name from the header record is entered into ESTAB, with value given by CSADDR.
- All external symbols appearing in the DEFINE records for the current control section are also entered into ESTAB.

- Their addresses are obtained by adding the value (offset) specified in the DEFINE to CSADDR.
- At the end, ESTAB contains all external symbols defined in the set of control sections together with the addresses assigned to each.
- A Load Map can be generated to show these symbols and their addresses.

A Load Map

Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
   begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag {duplicate external symbol}
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end {for}
         end {while ≠ 'E'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Algorithm - PASS 2

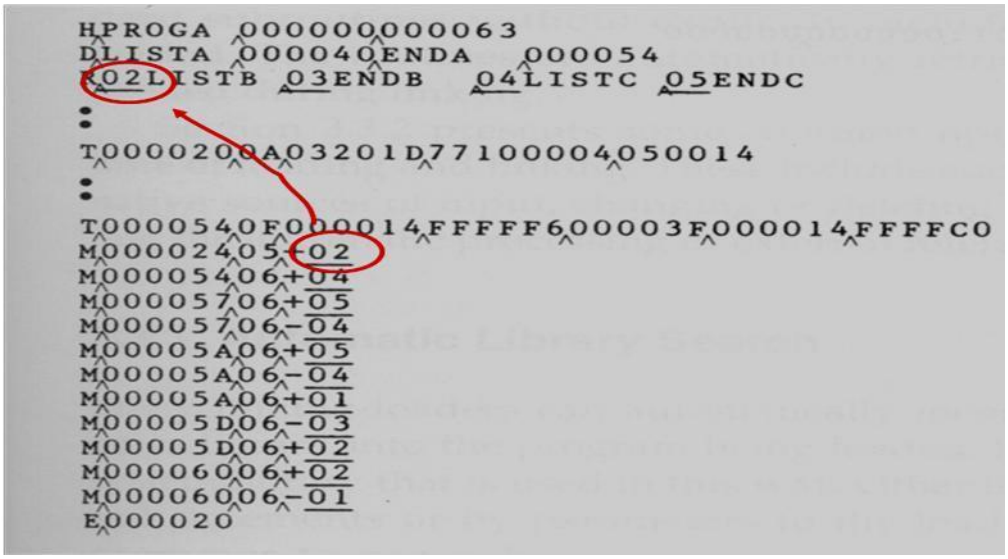
- During pass 2, the loader performs the actual loading, relocation, and linking.
- CSADDR is used in the same way as it was used in pass 1
  - It always contains the actual starting address of the control section being loaded.
- As each text record is read, the object code is moved to the specified address (plus CSADDR).
- When a modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.
- This value is then added to or subtracted from the indicated location in memory.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record {Header record}
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end  {if 'M'}
         end {while ≠ 'E'}
      if an address is specified {in End record} then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Reference Number

- The linking loader algorithm can be made more efficient if we assign a reference number to each external symbol referred to in a control section.
- This reference number is used (instead of the symbol name) in modification record.
- These simple techniques avoid multiple searches of ESTAB for the same symbol during the loading of a control section.
  - After the first search for a symbol (the REFER records), we put the found entries into an array.
  - Later in the same control section, we can just use the reference number as an index into thearray to quickly fetch a symbol's value.



5. Explain the machine independent loader features. (11 marks) (MAY, NOV 2012)(NOV 2013)

Machine-independent Loader Features are

- Automatic Library Search
- Loader Options

Automatic Library Search:

- The use of automatic library search process for handling external references. This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking.
- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- In most cases there is a standard system library. (E.g., the standard C library)
- Other libraries may be specified by control statements or by parameters to the loader.

- This feature allows the programmer to use subroutines from one or more libraries(mathematical or statistical routines) are a part of the programming language.
- The subroutines called by the program are automatically fetched from the library, linked with the main program, and loaded.
- The programmer does not need to take any action beyond mentioning the subroutine names as external references in the source program. On some systems, this feature is referred to as automatic library call.
- Linking loader that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.
- At the end of pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
- The loader searches the library for routines that contain the definitions of these symbols, and processes the subroutines found by this search process exactly as if they had been part of the primary

  input stream.
- The subroutines fetched from a library in this way may themselves contain external references. It is necessary to repeat the library search process until all references are resolved.
- If unresolved references remain after the library search is completed, they are treated as errors.
- If a symbol (or a subroutine name) is defined both in the source program and in the library, the one in the source program is used first.
- A programmer can make his own library easily on UNIX by using the “ar” command.

Loader Options:

- Many loaders allow the user to specify options that modify the standard processing.
- It include such capabilities as a specifying alternative sources of input, changing or deleting external references and controlling the automatic processing of external references.
- The loader option allows the selection of alternative sources of input.

For example, the command

- **INCLUDE program-name (library name)**

might direct the loader to read the designated object program from a library program from a library.

- **DELETEcsect-name**

might instruct the loader to delete the named control sections from the set of programs being loaded.

- **CHANGE name1, name2**

might cause the external symbol name1 to be changed to name2 wherever it appears in the object program.

The primary loader input with the loader commands

LIBRARY UTLIB

INCLUDE READ (UTLIB)

INCLUDE WRITE (UTLIB)

DELETE RDREC, WRREC

CHANGE RDREC, READ

CHANGE WRREC, WRITE

NOCALL STDDEV, PLOT, CORREL

- The commands are, use UTLIB (utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ.
- Similarly a reference to WRREC is changed to WRITE.
- Finally, no call to the functions SQRT, PLOT, if they are used in the program.

6. Explain the different options of loader design. (11 marks) (NOV 2012, 2013)

The different loader design options are

i. Linkage Editors

ii. Dynamic Linking

iii. Bootstrap Loaders

(i) LINKAGE EDITORS

- A linking loader performs all linking and relocation operations including automatic library search and loads the linked program directly into memory for execution.
- A ***linkage editor*** produces a linked version of the program (often called a ***load module*** or an ***executable image***), which is normally written to a file for later execution.
- When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory.
- The only object code modification necessary is the addition of an actual address to relative values within the program.
- The linkage editor performs relocation of all control sections relative to the start of the linked program.
- All items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required.

- If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.
- Resolution of external references and library searching are only performed once.
- A linking loader searches libraries and resolves external references every time the program is executed.

Advantages

- Linkage editors can perform many useful functions besides simply preparing an object program for execution.
- For example a program (PLANNER) that uses a large number of subroutines.
- Suppose that one sub-routine (PROJECT) used by the program (PLANNER) is changed to correct an error or to improve efficiency.
- After the new version of PROJECT is assembled or compiled, the linkage editor can be used to replace this subroutine in the linked version of the PLANNER.

The linkage editor commands.

```
                PLANNER
INCLUDE         (PROGLIB)
                                  {DELETE from existing
DELETE          PROJECT           PLANNER}
INCLUDE         PROJECT (NEWLIB)  {INCLUDE new version}
                PLANNER
REPLACE         (PROGLIB)
```

- Linkage editors can be used to build packages of subroutines or other control sections that are generally used together.
- This can be useful when dealing with subroutines libraries that support high level programming languages.

The linkage editor also used to combine the appropriate subroutines into a package with a FORTRAN command sequence like

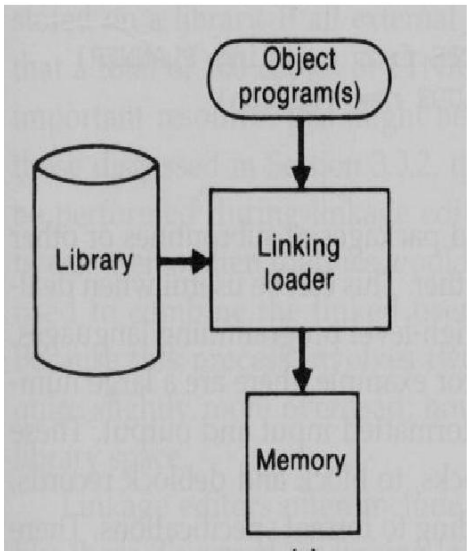
```
INCLUDE READR (FTNLIB)
                                  Data blocks
INCLUDE WRITER (FTNLIB)

INCLUDE BLOCK (FTNLIB)
                                  Records
INCLUDE DEBLOCK (FTNLIB)

INCLUDE ENCODE (FTNLIB)
                                  Data items
INCLUDE DECODE (FTNLIB)
.
.
SAVE FTNIO (SUBLIB)
```

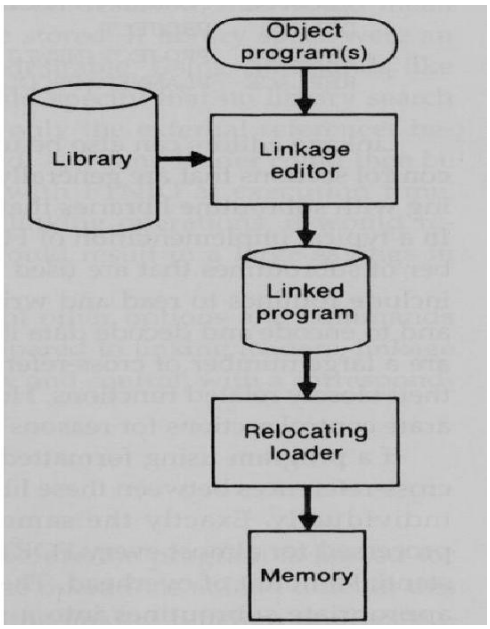
The difference between a Linkage Editor and a Linking Loader

Linking Loader	Linkage Editor
A linking loader performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution.	A linkage editor produces a linked version of the program, which is normally written to a file for later execution.
Linkage editors perform linking operations before the program is loaded for execution.	Linking loaders perform these same operations at load time.

Processing of an object program using

a) Linking Loader **b) Linkage Editor**


Object program(s)
Library
Linking loader
Memory


Object program(s)
Library
Linkage editor
Linked program
Relocating loader
Memory

(ii) DYNAMIC LINKING

Dynamic linking postpones the linking function until execution time; a subroutine is loaded and linked to the test of the program when it is first called. This type of function is usually called ***dynamic linking,dynamic loading or load on call.***

Dynamic linking applications

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.
- For example, run-time support routines for a high level language like C could be stored in ***a dynamiclink library.***
- A single copy of the routines in this library could be loaded into memory.
- In an object-oriented system, dynamic linking is often used for references to software objects.
- This allows the implementation of the object and its method to be determined at the time the program is run. (e.g., C++)
- This implementation can be changed at any time, without affecting the program that makes use of the object.
- Dynamic linking also makes it possible for one object to be shared by several programs.

Dynamic Linking Advantage

- The program contains subroutines that correct or diagnose errors in the input data during execution.
- However, the program were completely linked before execution, these subroutines must be loaded and linked every time the program is run.
- Dynamic linking provides the ability to load the routines only when they are needed. If the subroutines involved large or many external references, it can save both time and memory space.

Dynamic Linking Implementation

- The number of different mechanisms that can be used to accomplish the actual loading and linking called subroutine.
- A subroutine that is to be dynamically loaded must be called via an operating system service request.
- This method can also be thought of as a request to a part of the loader that is kept in memory during execution
- n of the program.

(a). Instead of executing a JSUB instruction to an external symbol, the program makes a load-and-call servicerequest to the OS.The parameter of this request is the symbolic name of the routine to be called.

(b). The OS examines its internal tables to determine whether the subroutine is already loaded. If needed, thesubroutine from the specified user or system libraries.

(c). Control is passed from the OS to the subroutine being called.

(d). When the called subroutine completes its processing, it returns to its caller (operating system).The OSthen returns control to the program that issues the request.

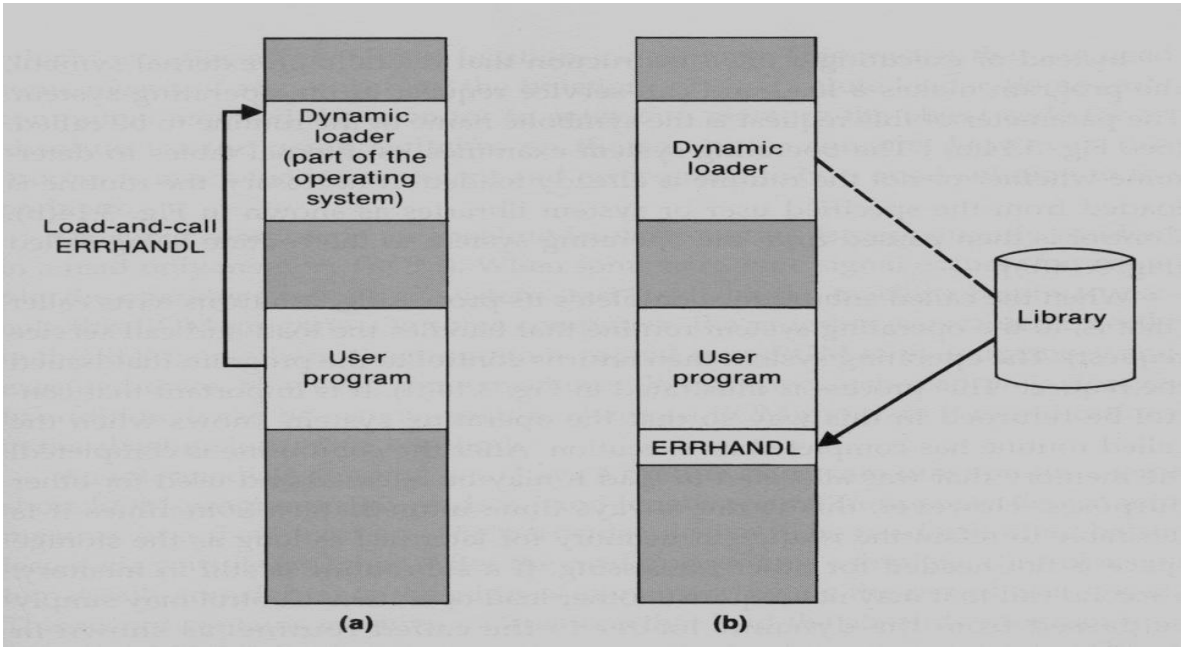
- After the subroutine is completed, the memory that was allocated to it may be released.
- However, often this is not always done immediately. If the subroutine is retained in memory, it can be used by later calls to the same subroutine without loading the same subroutine multiple times.

(e). Control can simply be passed from the dynamic loader to the called routine directly.

Implementation Example:

Issue a Load and Call service request

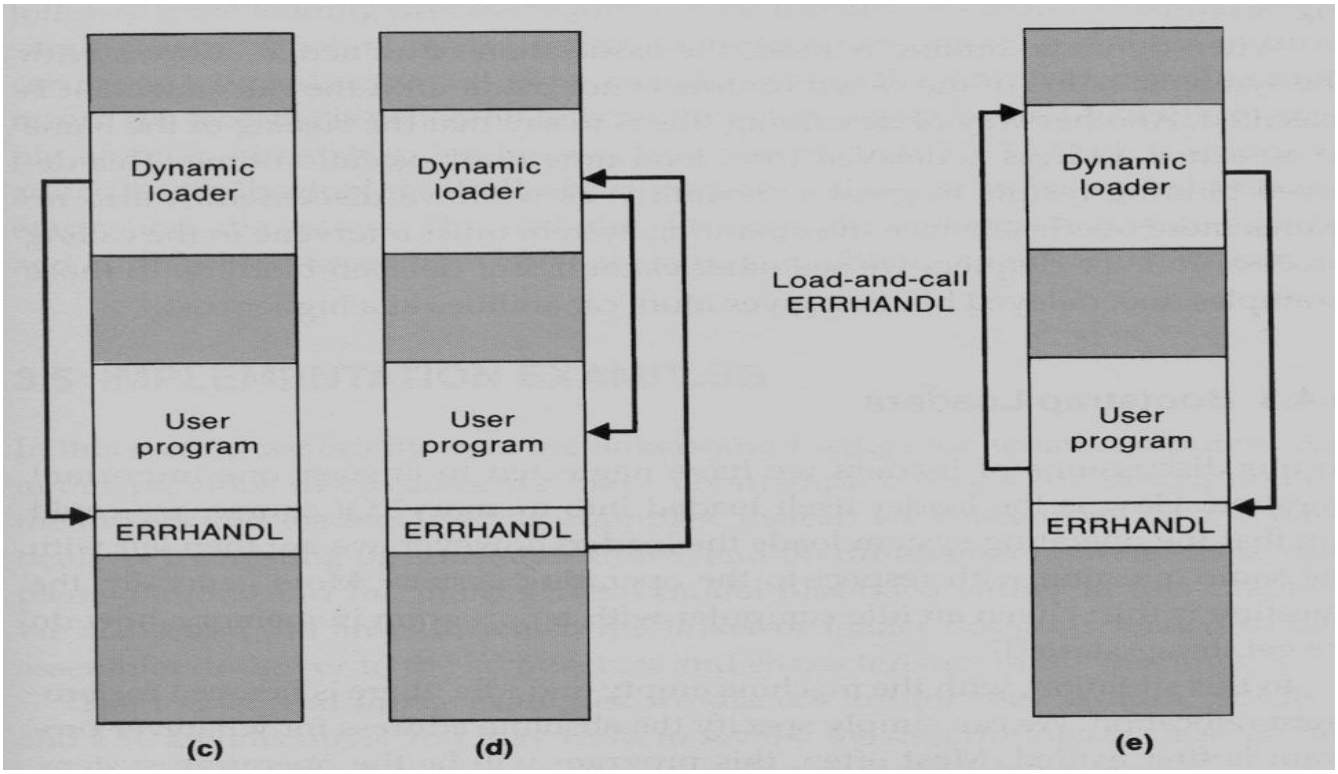
Load the called subroutine into memory



Control is passed to the loaded Subroutine

Control is returned to the loader and later returned to the user

The called subroutine this time is already loaded program



(iii) Bootstrap loaders

- The operating system loads the loader. When computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or a Bootstrap loader, which in turn loads OS and prepares it for execution.
- The first record (or records) is generally referred to as a ***bootstrap loader*** – makes the OS to be loaded.
- Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

PONDICHERRY UNIVERSITY QUESTIONS

2 MARKS

1. What is address sensitive program? **(NOV 2011) (Ref.Qn.No.11, Pg.no.4)**
2. List out the functions performed by relocating loaders? **(NOV 2011) (Ref.Qn.No.13, Pg.no.4)**
3. What is Bootstrap Loader? **(MAY 2012) (Ref.Qn.No.9, Pg.no.3)**
4. What is an absolute loader? State its disadvantages. **(MAY 2013) (Ref.Qn.No.6, Pg.no.2)**
5. Why is linking required after a program is translated? Define link editor. **(MAY 2013) (Ref.Qn.No.27,28 Pg.no.6)**
6. What are the two primary tasks of a linker? **(NOV 2013) (Ref.Qn.No.3, Pg.no.2)**
7. How are duplicate literal operands handled in machine independent loader? **(NOV 2013) (Ref.Qn.No.21, Pg.no.5)**

11 MARKS

NOV 2011(REGULAR)

1. Discuss the functions and design of an absolute loader. **(Ref.Qn.No.2, Pg.no.10)**

   (OR)

2. (a) What is dynamic linking? Explain (6) **(Ref.Qn.No.6, Pg.no.29)**

   (b) Write about bootstrap loaders. (5) **(Ref.Qn.No.3, Pg.no.12)**

MAY 2012(ARREAR)

1. Discuss about Basic Loader Functions. **(Ref.Qn.No.2, Pg.no.10)**

   (OR)

2. Explain the machine independent loader features. **(Ref.Qn.No.5, Pg.no.24)**

NOV 2012(REGULAR)

1. Discuss about Machine independents loader features. **(Ref.Qn.No.5, Pg.no.24)**

   (OR)

2. Explain the different options of loader design. **(Ref.Qn.No.6, Pg.no.26)**

MAY 2013(ARREAR)

1. Define bootstrap loader and state the algorithm for the same. **(Ref.Qn.No.3, Pg.no.12)**

   (OR)

2. Explain with neat block diagram the role of loader and linker. **(Ref.Qn.No.1, Pg.no.8)**

NOV 2013 (REGULAR)

1. Explain the loader design options. **(Ref.Qn.No.6, Pg.no.26)**

   (OR)

2. (a) Discuss the machine independent loader features in detail.(6) **(Ref.Qn.No.5, Pg.no.24)**

   (b) State and describe the significance of dynamic linking. (5) **(Ref.Qn.No.6, Pg.no.29)**

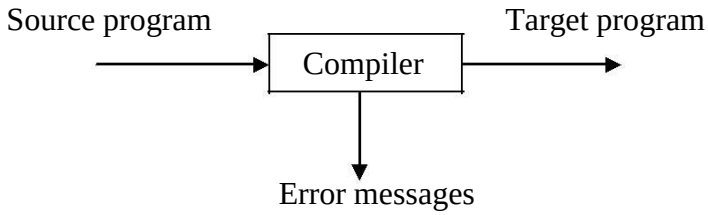
UNIT –III

SOURCE PROGRAM ANALYSIS

2 MARKS

1. What is a compiler? (MAY, NOV 2012)

A *compiler* is a program that reads a program written in one language – the source language and translates it into an equivalent program in another language – the target language. In translation process, the compiler reports to its user the presence of errors in the source program.



2. What are the classifications of a compiler?

Compilers are sometimes classified as:

- Single-pass
- Multi-pass
- Load-and-go
- Debugging or
- Optimizing

3. What are the two parts of a compilation? Explain briefly.

There are two parts to compilation as

1. Analysis part
2. Synthesis part

- The *analysis part* breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The *synthesis part* constructs the desired target program from the intermediate representation.

4. What are the tools available in analysis phase?

Many software tools that manipulate source programs are

- Structure editors
- Pretty printers
- Static checkers
- Interpreters

5. What is Query Interpreters?

A *Query interpreter* translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

6. List the analysis of the source program?

Analysis consists of three phases:

- Linear Analysis.
- Hierarchical Analysis.
- Semantic Analysis.

7. What is linear analysis?

- In a compiler, *linear analysis* is called *lexical analysis* or *scanning.*
- *Linear analysis* in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

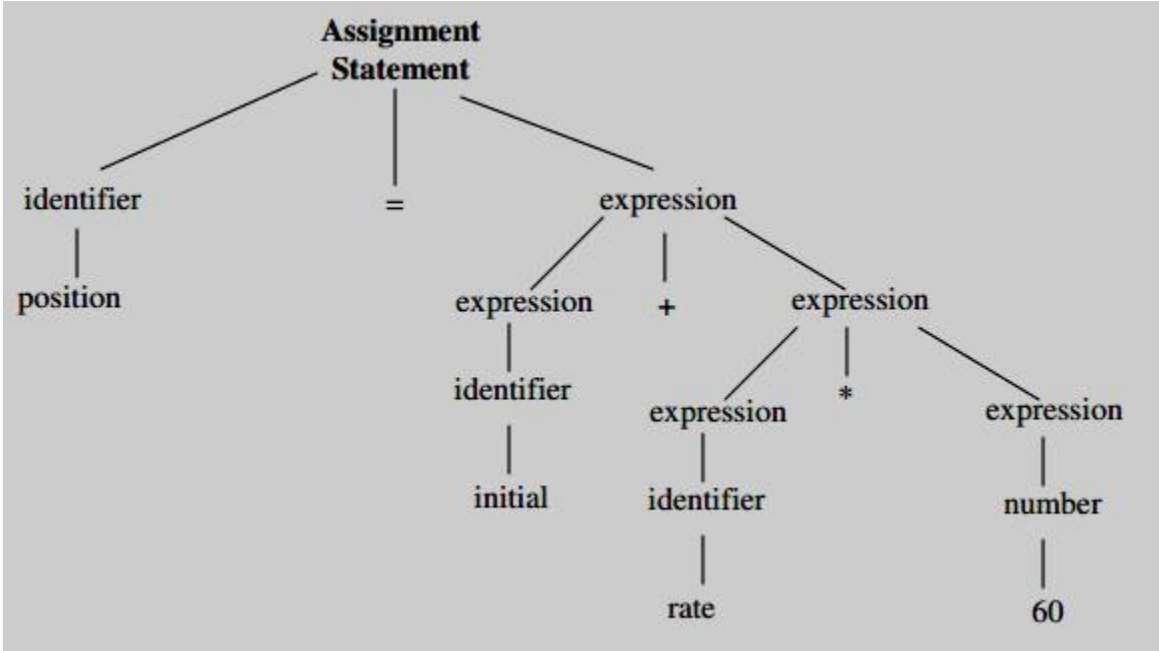
8. What is hierarchical analysis? (NOV 2011)

- *Hierarchical analysis* is called *parsing* or *syntax analysis*.
- *Hierarchical analysis* involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.

9. What is semantic analysis?

- The *Semantic analysis*, it checks the source program for semantic errors and gathers type information for the subsequent code generation phase.
- It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.

10. Draw the parse tree for a source program as position: = initial + rate * 60.



11. List the various phases of a compiler.

The following are the various phases of a compiler:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate code generator
- Code optimizer
- Code generator

12. What is a symbol table?

- A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store

  or retrieve data from that record quickly.

- Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

13. What is Intermediate code generator?

The intermediate representation should have two important properties;

- it should be easy to produce, and
- it should be easy to translate into the target program.

14. What is three address code?

- An intermediate form called "*three-address code*, "which is like the assembly language for a machine in which every memory location can act like a register.
- Three-address code consists of a sequence of instructions, each of which has at most three operands.

15. What is code generator?

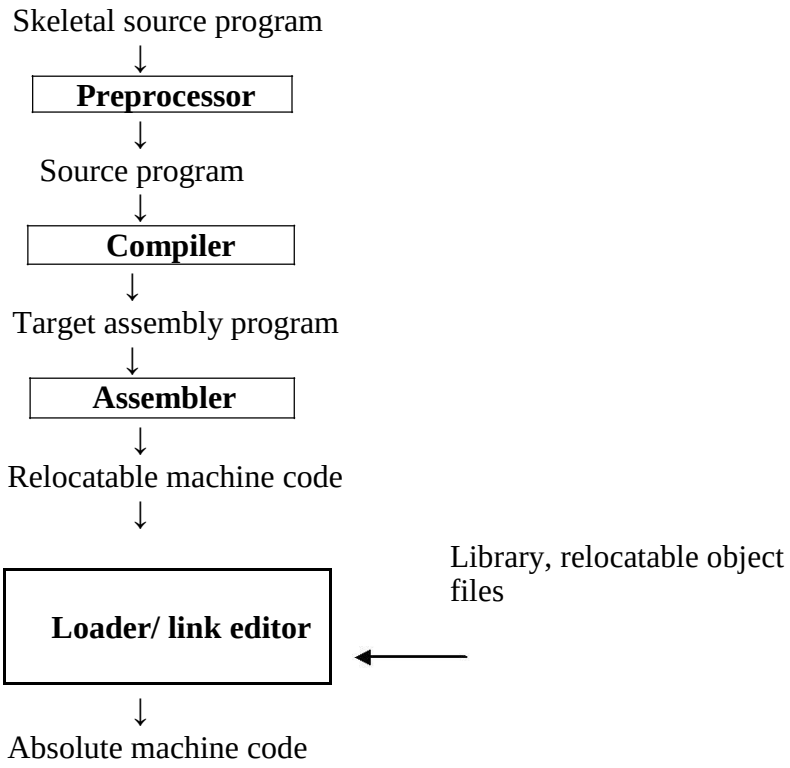
- The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

16. Mention the cousins of the compiler?

Cousins of the compiler are:

- Preprocessor
- Compiler
- Assembler
- Two-Pass Assembly
- Loader and Link-Editor

17. State with example the cousins of compilers. (NOV 2013)



18. What is Preprocessors? List its functions.

Preprocessors produce input to compilers. They may perform the following functions:

- Macro Processing
- File inclusion
- Rational Preprocessors
- Language extensions

19. Define Assembly code?

- Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.
- A typical sequence ***b := a + 2***,the assembly instructions might be

MOV a, R1

ADD #2, R1

MOV R1, b

20. What is the use Two-Pass assembly?

The assembler makes two passes over the input, where a *pass* consists of reading an input file once.

- In the *first pass*, all the identifiers that denote storage locations are found and stored in a symbol table.
- Identifiers are assigned storage locations as they are encountered for the first time.
- In the *second pass*, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into address given for that identifier in the symbol table.
- The output of the second pass is usually relocatable machine code.

21. What is loader and link-editor?

- Usually, a program called a loader performs the two functions of *loading* and *link-editing.*
- The process of *loading* consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper location.
- The *link-editor* allows us to make a single program from several files of relocatable machine code. These files may be the result of several different compilation and one or more library files of routines

  provided by the system.

22. State the function of front end and back end of a compiler phase. (MAY 2013)

The front end consists of those phases that depends primarily on the source language and are largely independent of the target machine.

These include

- Lexical analysis
- Syntactic analysis
- Semantic analysis
- Creation of symbol table
- Generation of intermediate code
- Code optimization
- Error handling

23. State the function back end of a compiler phase. (MAY 2013)

The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language.

These include

- Code optimization
- Code generation
- Error handling and
- Symbol-table operations

24. What is single pass?

Several phase of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.

25. Define compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as *compiler-compilers, compiler-generators or translator-writing systems.* Largely they are oriented around a particular model of languages, and they are suitable for generating compilers of languages similar model.

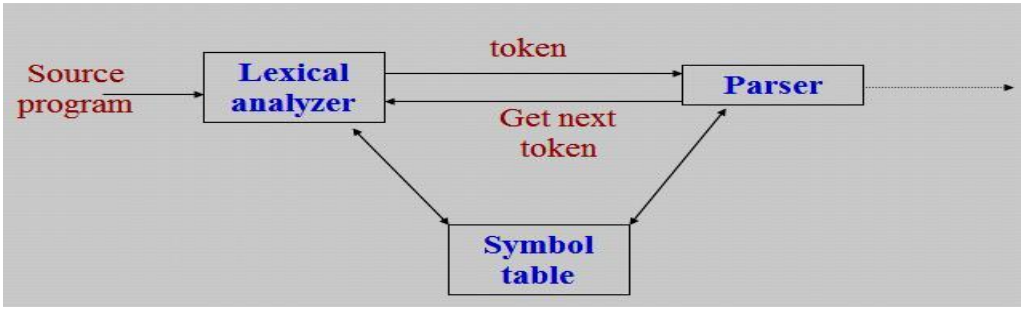
26. List the various compiler construction tools.

The various compiler construction tools are

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

27. What is the role of lexical analyzer? (NOV 2013)

- The *lexical analyzer* is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses the syntax analysis.
- Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.



28. What are the issues in lexical analysis?

The issues in lexical analysis are

- Simpler design is the most important consideration.
- Compiler efficiency is improved.
- Compiler portability is enhanced.

29. Why separate lexical analysis phase is required? (MAY 2013)

i. Simpler design is the most important consideration.
- Comments and white space have already been removed by lexical analyzer.

ii. Compiler Efficiency is improved.
- Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

iii. Compiler Portability is enhanced.
- Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

30. What is the major advantage of a lexical analyzer generator? (NOV 2011)

The major advantages of a lexical analyzer generator are

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

31. What are tokens, patterns, and lexeme?

- *Tokens*- Sequence of characters that have a collective meaning.
- *Patterns*- 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.
- *Lexeme*- A sequence of characters in the source program that is matched by the pattern for a token.

32. Differentiate between tokens, patterns, and lexeme?

Token	lexeme	patterns
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except “

33. List the various Panic mode recovery in lexical analyzer?

Possible error recovery actions are:

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

34. What are the approaches to implement lexical analyzer?

The three general approaches to the implementation of a lexical analyzer

1. Use a lexical analyzer generator such as the Lex compiler to produce a regular expression based specification. In this case, the generator provides for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems programming languages using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and reading of input.

35. What is input buffering?

The *input buffer* is useful when look-ahead on the input is to identify tokens.

36. Define sentinels.

- *Sentinel* is a special character that cannot be part of the source program.
- The techniques for speeding up the lexical analyzer, use the "sentinels " to mark the buffer end.

37. List the operations on languages.

The operations that can be applied to languages are

- Union - L U M ={s | s is in L or s is in M}
- Concatenation – LM ={st | s is in L and t is in M}
- Kleene Closure – L* (zero or more concatenations of L)
- Positive Closure – L+ (one or more concatenations of L)

38. Write a regular expression for an identifier.

- An identifier is defined as a letter followed by zero or more letters or digits. The regular expression for an identifier is given as

letter (letter | digit)*

39. How the regular expression can be defined in specification of the language. 1. □ is a regular expression that denotes {□}, the set containing empty string.

2. If *a* is a symbol in □, then *a* is a regular expression that denotes {*a*}, the set containing the string *a*. **3.** Suppose *r* and *s* are regular expressions denoting the language L(*r*) and L(*s*), then

- (*r*) |(*s*) is a regular expression denoting L(*r*) □L(*s*).
- (*r*)(*s*) is regular expression denoting L (*r*) L(*s*).
- (*r*) * is a regular expression denoting (L (*r*))*.
- (*r*) is a regular expression denoting L (*r*).

40. Mention the various notational short hands for representing regular expressions.

- One or more instances
- Zero or one instance
- Character classes
- Non regular sets

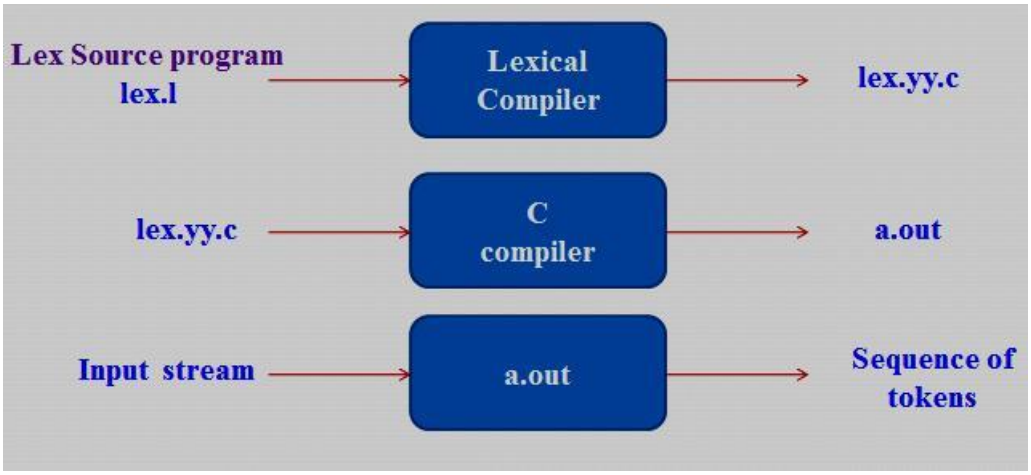
41. What is transition diagram? (NOV 2012)

- An intermediate step in the construction of a lexical analyzer, we first produce a stylized flowchart called a *transition diagram.*
- Transition diagram depicts the actions that take place when a lexical analyzer is called by the parser to get the next token.

42. Define Lex complier?

A particular tool called *Lex*, used to specify lexical analyzers for a variety of languages. We refer to the tool as the *Lex compiler* and to its input specification as the Lex language.

43. How to create a lexical analyzer with Lex?



44. List out the parts on Lex specifications. (MAY 2012)

A Lex program consists of three

parts: declarations

%%

translation rules

%%

auxiliary procedures

- The declarations section includes declarations of variables, manifest constants, and regular definitions.
- The translation rules of a Lex program are statement of the form

*p*1 {

action1} *p*2

{ *action2*}

…… …..

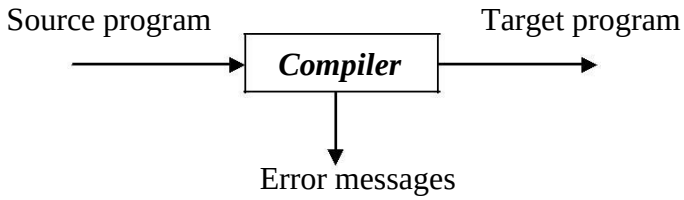
*p*n { *actionn*}

- The auxiliary procedures are needed by the actions. These procedures can be compiled separately and loaded with the lexical analyzer.

11 MARKS

1. Write a short note on compiler design? (6 marks)

A ***compiler*** is a program that reads a program written in one language – the source language and translates it into an equivalent program in another language – the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



- At first, the variety of compilers may appear overwhelming. There are thousands of source languages, ranging from traditional programming languages such as FORTRAN and Pascal to specialized languages in

  every area of computer application.
- Target languages are equally as varied; a target language may be another programming language, or the machine language of any computer between a microprocessor and a supercomputer.

Compilers are sometimes classified as:

- Single-pass
- Multi-pass
- Load-and-go
- Debugging or
- Optimizing

- Throughout the 1950's, compilers were considered notoriously difficult programs to write. The first FORTRAN compiler, for example, took 18 staff-years to implement.

Analysis-Synthesis Model of Compilation:

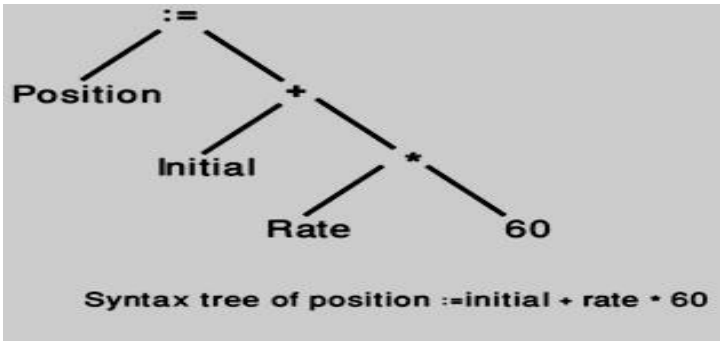
There are two parts to compilation as

1. Analysis part
2. Synthesis part

- The *analysis part* breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The *synthesis part* constructs the desired target program from the intermediate representation.

- During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a ***tree.***
- Often, a special kind of tree called a ***syntax tree*** is used, in syntax tree each node represents an operation and the children of the node represent the arguments of the operation.

For example, a syntax tree of an assignment statement is **position: = initial + rate * 60.**



Syntax tree of position :=initial + rate * 60

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

- Structure editors
- Pretty printers
- Static checkers
- Interpreters

Structure editors:

A *structure editor* takes as input a sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, appropriate hierarchical structure on the source program. It is useful in the preparation of source program.

Pretty printers:

A *pretty printer* analyzes a program and prints it in a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font.

Static checkers:

A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the source program.

Interpreters:

Interpreters are frequently used to execute command languages, since each operator executed in command languages is usually a complex routine such as an editor or compiler.

The analysis portion in each of the following examples is similar to that of a conventional compiler.

- Text formatters
- Silicon compilers
- Query interpreters

Text formatters:

A *text formatter* takes input that is a stream of characters, most of which is text to be typeset, and it includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and the superscripts.

Silicon compilers:

A *silicon compiler* has a source language that is similar or identical to a conventional programming language. However, the variables of the language represent, not locations in memory, but also logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language.

Query interpreters:

A *Query interpreter* translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

2. Explain the analysis of the source program? (11 marks)

In compiling, analysis consists of three phases:

- Linear Analysis
- Hierarchical Analysis
- Semantic Analysis

Linear Analysis:

- *Linear analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning
- In a compiler, linear analysis is called lexical ***analysis or scanning.***

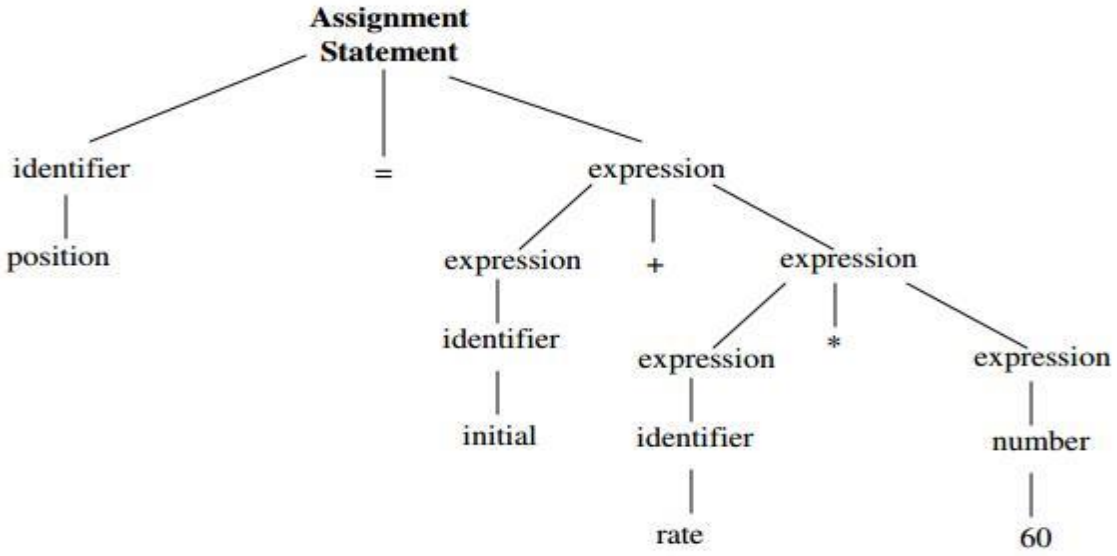
For example, in lexical analysis the characters in the assignment statement **position: = initial + rate * 60** would be grouped into the following tokens:

1. The identifier position.
2. The assignment symbol :=
3. The identifier initial.
4. The plus sign.
5. The identifier rate.
6. The multiplication sign.
7. The number 60.

- The blanks separating the characters of these tokens would normally be eliminated during the lexical analysis.

Hierarchical Analysis:

- *Hierarchical analysis* is called ***parsing* or *syntax analysis*.**
- Hierarchical analysis involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.
- The grammatical phrases of the source program are represented by a parse tree.



Parse tree for position: = initial + rate * 60

- In the expression **initial + rate * 60**, the phrase rate * 60 is a logical unit because the usual conventions of arithmetic expressions tell us that the multiplication is performed before addition.
- Because the expression **initial + rate** is followed by a *, it is not grouped into a single phrase by itself.

- The hierarchical structure of a program is usually expressed by *recursive rules*. For example, the following rules, as part of the definition of expression:

1. Any *identifier* is an expression.
2. Any *number* is an expression
3. If *expression1* and *expression2* are expressions, then so are
   - *expression1 + expression2*
   - *expression1 * expression2*
   - (*expression1*)

- Rules (1) and (2) are non-recursive basic rules, while (3) defines expressions in terms of operators applied to other expressions.

Similarly, many languages define statements recursively by rules such as: **1.** If *identifier1* is an identifier and *expression2* is an expression, then

identifier1 :=

expression2 is a statement.

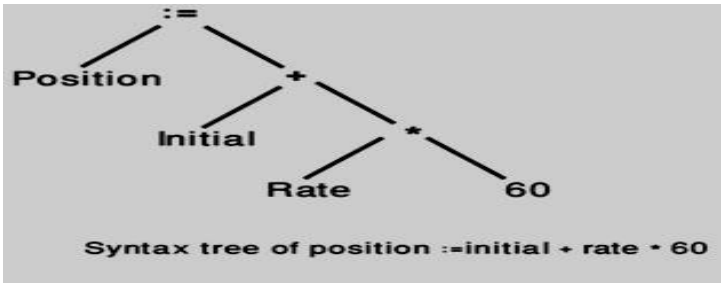
2. If *expression1* is an expression and *statement2* is a statement,

then ***while*** *(expression1)* ***do*** *statement2*

if *(expression1)* ***then***

statement2 are statements.

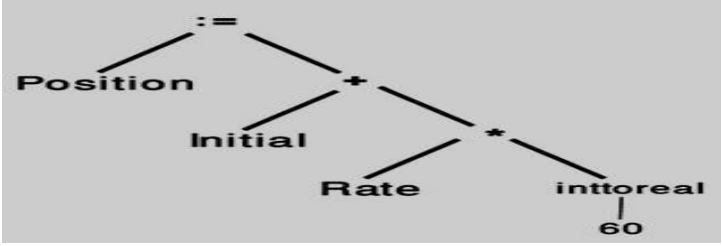
A ***syntax tree*** is a compressed representation of the parse tree in which the operators appear as the interior nodes and the operands of an operator are the children of the node for that operator.



Syntax tree of position :=initial + rate * 60

Semantic Analysis:

- The *semantic analysis* phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.
- An important component of semantic analysis is *type checking.*
- The compiler checks that each operator has operands that are permitted by the source language specification.
- For example, when a binary arithmetic operator is applied to an integer and real.
- In this case, the compiler may need to be converting the integer to a real. As shown in figure given below.



3. Explain the various phases of a compiler with an example? (11 marks) (NOV 2011, 2013) (MAY, NOV 2012)(MAY 2013)

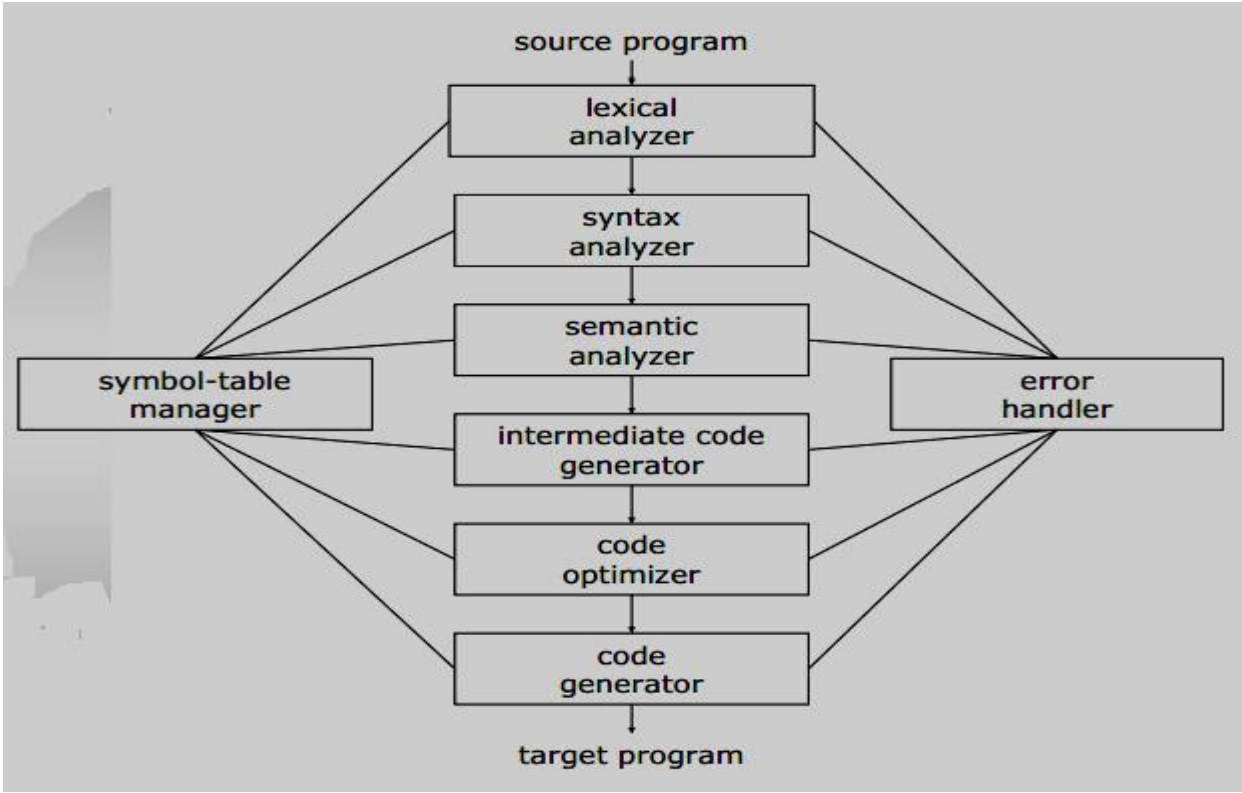
A compiler operates in phases, each of which transforms the source program from one representation to another. The six phases of a complier are

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate code generator
5. Code optimizer
6. Code generator

Two other activities are

- Symbol table Manager
- Error handler

A typical decomposition of a compiler is shown in fig given below



Phases of a compiler

The Analysis Phases:

As translation progresses, the compiler's internal representation of the source program changes. Consider the translation of the statement,

position: = initial + rate * 10

Lexical analyzer:

- The ***lexical analysis*** phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an

- identifier, a keyword (if, while, etc.), a punctuation character or a multi-character operator like :=. In a compiler, *linear analysis* is called ***lexical analysis* or *scanning.***

- ***Linear analysis*** in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters.
- The character sequence forming a token is called the ***lexeme*** for the token.

- Certain tokens will be augmented by a '*lexical value'*. For example, when an identifier like *rate* is found, the lexical analyzer not only generates a token id, but also enters the lexeme rate into the symbol table,

  if it is not already there.

Consider **id1, id2** and **id3** for **position, initial, and rate** respectively, that the internal representation of an identifier is different from the character sequence forming the identifier.

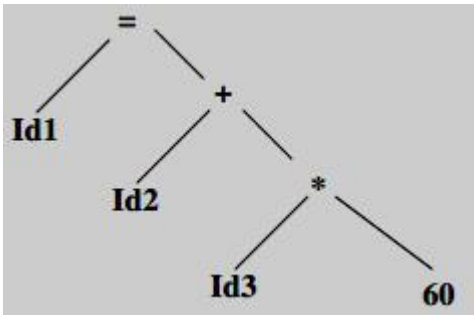
The representation of the statement given above after the lexical analysis would

be: **id1: = id2 + id3 * 10**

Syntax analyzer:

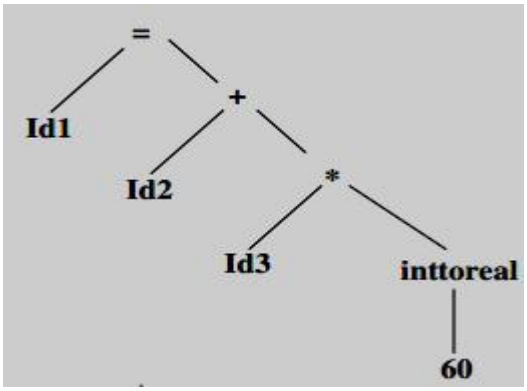
- ***Hierarchical analysis*** involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.
- *Hierarchical analysis* is called ***parsing* or *syntax analysis.***

- Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax trees.



Semantic analyzer:

- The ***Semantic analysis,*** it checks the source program for semantic errors and gathers type information for the subsequent code generation phase.
- It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.
- Compiler report an error, if **real number** is used to **index** an array.
- The bit pattern representing an integer is generally different from the bit pattern for a real, even they have the same value.
- For example, the identifiers position, initial, rate declared to be **real** and that 60 by itself assumed to be **integer.**
- The general approach is to convert the integer to a real.



Intermediate code generator:

- After Syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. The intermediate representation as a program for an abstract machine.
- This intermediate representation should have two important properties;
  - it should be easy to produce, and
  - it should be easy to translate into the target program.
- An intermediate form called "*three-address code*, "which is like the assembly language for a machine in which every memory location can act like a register.
- Three-address code consists of a sequence of instructions, each of which has at most three operands.
- The source program might appear in three-address code as

temp1: = inttoreal (60)

temp2: = id3 * temp1

temp3: = id2 + temp2

id1: = temp3

This intermediate form has several properties:

1. First, each three address instruction has at most one operator in addition to the assignment. Thus, when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program.
2. Second, the compiler must generate a temporary name to hold the value computed by each instruction.
3. Third, some "three-address" instructions have fewer than three operands.

Code Optimization:

- The *code optimization phase* attempts to improve the intermediate code, so that faster-running machine code will result.
- For example, a natural algorithm generates the intermediate code, using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions.

  temp1 := id3 * 60.0

  id1 := id2 + temp1

- There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase.
- That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated.
- There is a great variation in the amount of code optimization different compilers.
- 'Optimizing compilers', a significant fraction of the time of the compiler is spent on this phase.

Code Generation:

- The final phase of the compiler is the *generation of target code*, consisting normally of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.
- The assignment of variables to registers.

For example, using registers 1 and 2, the translation of the code as

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

- The first and second operands of each instruction specify a source and destination, respectively.
- The ***F*** in each instruction tells us that instructions deal with floating-point numbers.
- This code moves the contents of the address id3 into register 2, and then multiplies it with the real-constant 60.0.
- The **#** signifies that 60.0 is to be treated as a constant.
- The third instruction moves id2 into register 1 and adds to it the value previously computed in register 2.
- Finally, the value in register 1 is moved into the address of id1.

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 identifier.
- These attributes may provide information about the storage allocated for an identifier, its type, its scope and in case of procedure names, such things at the number and types of its arguments and methods of passing each argument and type returned.
- The ***symbol table*** is a data structure containing a record for each identifier with fields for the attributes of the identifier.
- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.
- Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.
- However, the attributes of an identifier cannot normally be determined during lexical analysis. For example, in a ***Pascal declaration*** like

  var position, initial, rate : real;
- The type real is not known when position, initial and rate are seen by the lexical analyzer.
- The remaining phases get information about identifiers into the symbol table and then use this information in various ways.

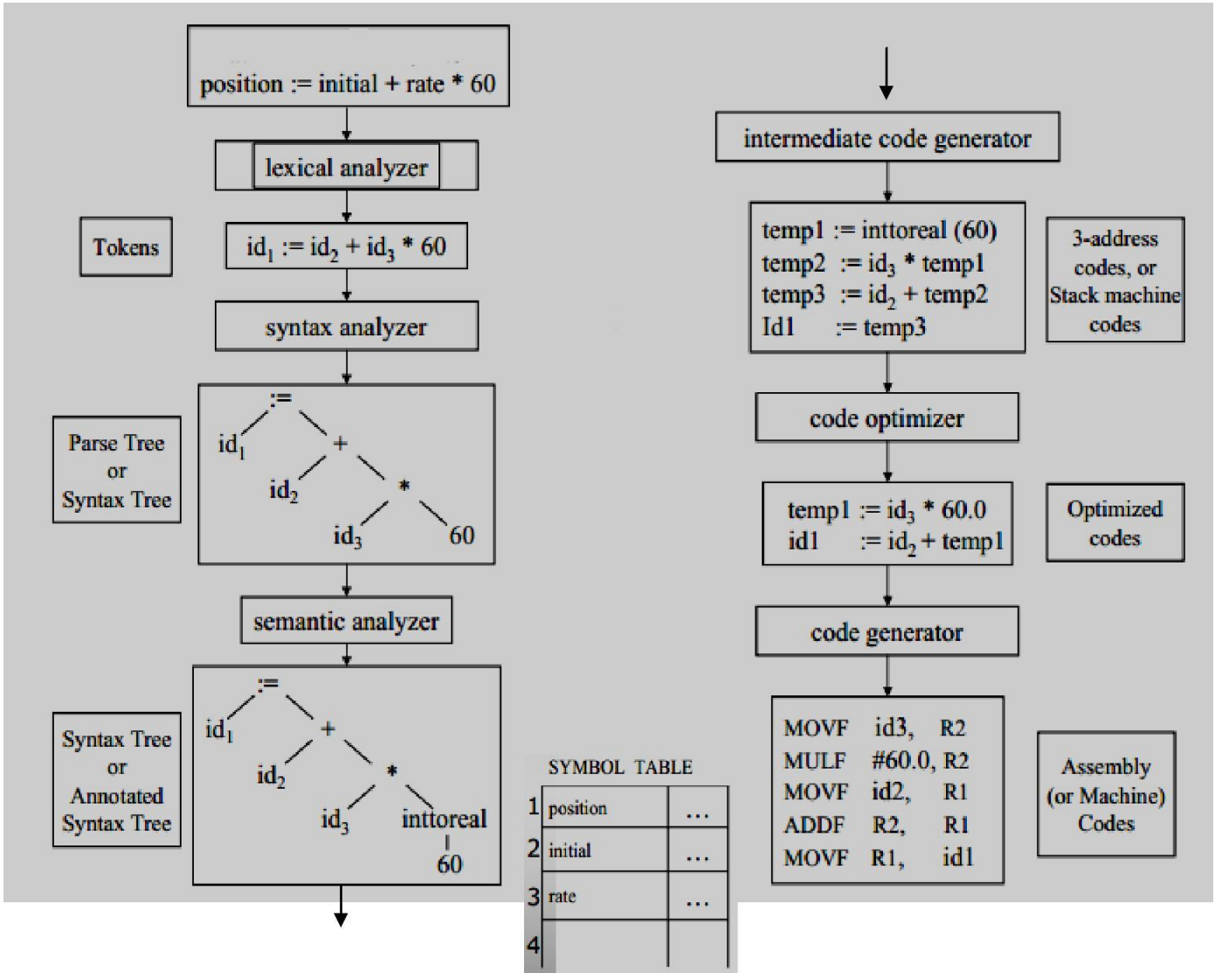
Error Detection and Reporting:

- Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- A compiler that stops when it finds the first error is not as helpful as it could be.
- The *syntax and semantic analysis* phases usually handle a large fraction of the errors detectable by the compiler.

- The *lexical phase* can detect errors where the characters remaining in the input do not form any token of the language.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
- During semantic analysis the compiler tries to detect the right syntactic structure but no meaning to the operation involved.
- e.g. we try to add two identifiers, one of which is the name of an array and the other the name of the procedure.

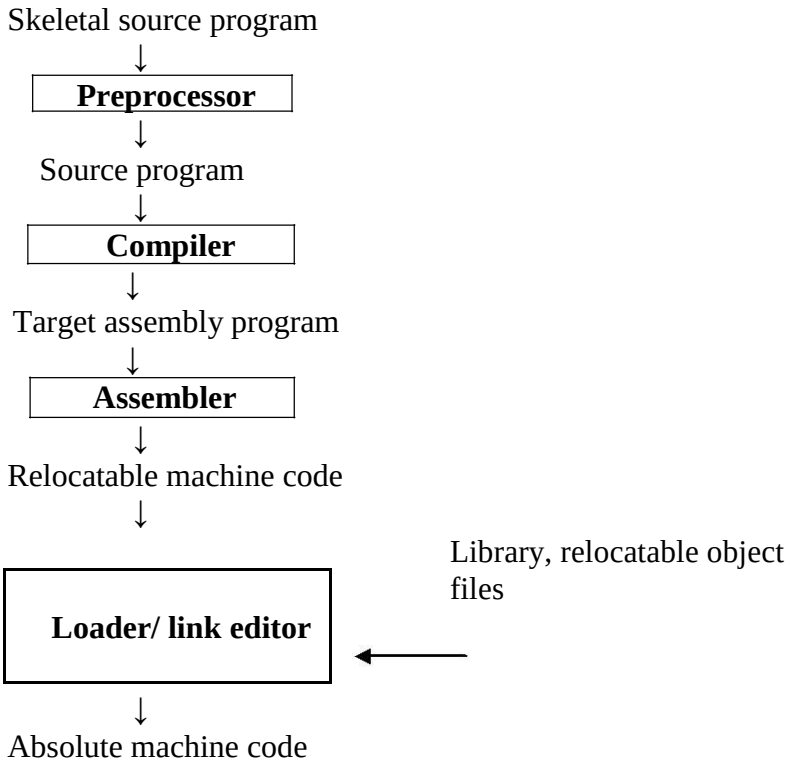
Example:

Consider the translation of the statement, **position: = initial + rate * 10**



4. Explain the Cousins of the compiler. (11 marks) (MAY 2012)

The input to a compiler may be produced by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained.



Preprocessors:

Preprocessors produce input to compilers. They may perform the following functions:

- Macro Processing:
- File inclusion:
- "Rational" Preprocessors:
- Language extensions:

Macro Processing:

- A preprocessor may allow a user to define macros that are shorthand's for longer constructs.

File inclusion:

- A preprocessor may include header files into the program text.
- *For example*, the C preprocessor causes the contents of the file ***<global.h>*** to replace the statement ***#include <global.h>*** when it processes a file containing this statement.

"Rational" Preprocessors:

- These processors augment older languages with more modern flow-of-control and data-structuring facilities.
- *For example*, such a preprocessor might provide the user with built-in macros for constructs like while statements or if statements.

Language extensions:

- These processors attempt to add capabilities to the language by what amounts to built-in macros.
- *For example*, the language Equal is a database query language embedded in C. Statements beginning with ## are taken by the preprocessor to be database-access statements, unrelated to C, and are translated into procedure calls on routines that perform the database access.

Macro processors deal with two kinds of statements:

1. Macro definition and
2. Macro use

- Definitions are normally indicated by unique character or keyword like define or macro. They consist of a name for the macro being defined and a body, forming its definition.
- The use of macro consists of naming the macro and supply actual parameters, (i.e.) values for its formal parameters.

Assemblers:

- Some compilers produce assembly code that is passed to an assembler for further processing.
- Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.
- *Assembly code* is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.
- A typical sequence ***b := a + 2***, the assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

- This code moves the contents of the address **a** into register 1, then adds the constant **2** to it, treating the contents of register 1 as a fixed-point number, and finally stores the result in the location named by **b**. thus, it computes b:=a+2.

Two - Pass Assembly:

The simplest form of assembler makes two passes over the input, where a *pass* consists of reading an input file once.

- In the *first pass*, all the identifiers that denote storage locations are found and stored in a symbol table.

  Identifiers are assigned storage locations as they are encountered for the first time.

Identifiers	Address
a	0
b	4

- In the *second pass*, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into address given for that identifier in the symbol table.
- The output of the second pass is usually *relocatable* machine code, that it can be loaded starting at any location L in memory.

Loaders and Link-Editors:

- Usually, a program called a *loader* performs the two functions of *loading* and *link-editing.*
- The process of *loading* consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper location.
- The *link-editor* allows us to make a single program from several files of relocatable machine code.
- These files may be the result of several different compilation and one or more library files of routines provided by the system.
- If the files are to be used together, there may be external references, in which the code of one file refers to a location in another file.

5. Write a short note on grouping of phases? (5 marks)

In an implementation, activities from more than one phase are often grouped together.

Front and Back Ends:

- The phases are collected into a *front end* and a *back end.*
- The ***front end*** consists of those phases that depend primarily on the source language and are largely independent of the target machine.

These normally include

- lexical analysis
- syntactic analysis
- semantic analysis
- the creating of the symbol table
- the generation of intermediate code.
- code optimization can be done by the front end as well.

- The front end also includes the error handling that goes along with each of these phases.

- The ***back end*** of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language.

These normally include

- Code optimization
- Code generation
- Error handling and
- Symbol-table operations

Passes:

- Several phase of compilation are usually implemented in a single ***pass*** consisting of reading an input file and writing an output file.
- For several phases to be grouped into one pass and for the activity of these phases to be interleaved during the pass.
- For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.
- The token stream after lexical analysis may be translated directly into intermediate code.

Reducing the Number of Passes:

- It is desirable to have relatively few passes, since it takes time to read and write intermediate files. If we group several phases into one pass, it may be forced to keep the entire program in memory, because one phase may need information in a different order then a previous phase produces it.
- The grouping into one pass presents few problems.
- The interface between lexical and syntax analyzers can often be limited to a single token.
- It is very hard to perform code generation until the intermediate representation has been completely generated.

6. State the different compiler construction tools and their use. (6 marks) (MAY 2013)

The compiler writer, like any programmer, can profitably use tools such as debuggers, version managers, profilers and so on.

- In addition to these software-development tools, other more specialized tools have been developed for helping implement various phases of a compiler.
- The first compilers were written; systems to help with the compiler-writing process appeared.
- These systems have often been referred to as
  - ***Compiler-compilers,***
  - ***Compiler-generators, or***
  - ***Translator-writing systems.***
- The general tools have been created for the automatic design of specific compiler components.

- These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of a compiler.

The various compiler construction tools are

1. Parser generators
2. Scanner generators
3. Syntax-directed translation engines
4. Automatic code generators
5. Data-flow engines

Parser generators:

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.
- This phase is considered one of the easiest to implement.

Scanner generators:

- These tools automatically generate lexical analyzers, normally from a specification based on regular expressions.
- The basic organization of the resulting lexical analyzer is in effect a finite automaton.

Syntax directed translation engines:

- These produce collections of routines that walk the parse tree, generating intermediate code.
- The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

Automatic code generators:

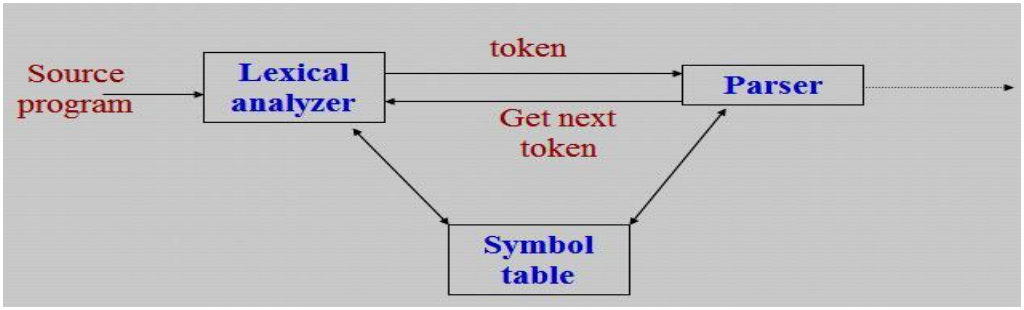
- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
- The rules must handle the different possible access methods for data.
- Eg; variables may be in registers, in a fixed location in memory or may be allocated a position on a stack. The basic technique is *"**template matching".***

Data-flow engines:

- Much of the information needed to perform good code optimization involves "data-flow analysis," the gathering of information how values are transmitted from one part of a program to each other part.

7. Discuss the role of the lexical analyzer. (11 marks) (NOV 2012)

- The *lexical analyzer* is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses the syntax analysis.
- This is implemented by making the lexical analyzer be a sub-routine or a co-routine 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.



Interaction of lexical analyzer with parser

The lexical analyzers is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface.

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

The lexical analyzers are divided into a cascade two phases are

1. *Scanning* is responsible for doing simple tasks.
2. *Lexical analysis* more complex operations

For example, a FORTRAN compiler might use a scanner to eliminate blanks from the input.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

i. Simpler design is the most important consideration.
- Comments and white space have already been removed by lexical analyzer.

ii. Compiler Efficiency is improved.
- A large amount of time is spent reading the source program and partitioning it into tokens.
- Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

iii. Compiler Portability is enhanced.

- Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.
- The representation of a special or non-standard symbol such as ↑ in Pascal can be isolated in the lexical analyzer.

Tokens, Patterns, and Lexemes:

- *Tokens*- Sequence of characters that have a collective meaning.
- *Patterns*- 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.
- *Lexeme*- A sequence of characters in the source program that is matched by the pattern for a token.

Token	lexeme	patterns
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except “

Attributes for Tokens:

- When more than one pattern matches a lexeme, the lexical analyzer must provide information about the particular lexeme that matched to the phases of a compiler.
- For example, the pattern **num** matches both the strings 0 and 1.
- The lexical analyzer collects information about tokens into their associated attributes.
- The *tokens* influence *parsing decisions*; the *attributes* influence the *translation of tokens.*
- A token has usually only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.

The tokens and associated attribute-values for the FORTRAN statement

E = M * C ** 2

<id, pointer to symbol-table entry for

R> <assign_op, >

<id, pointer to symbol-table entry for

M> <mult_op, >

<id, pointer to symbol-table entry for

C> <exp_op, >

<num, integer value 2>

Lexical Errors:

Possible error recovery actions or Panic mode recovery are

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

8. Explain the input buffering with sentinels. (6 marks) (NOV 2013)

- The two- buffer input scheme is useful when look-ahead on the input is necessary to identify tokens.
- The techniques for speeding up the lexical analyzer, use the "sentinels " to mark the buffer end.

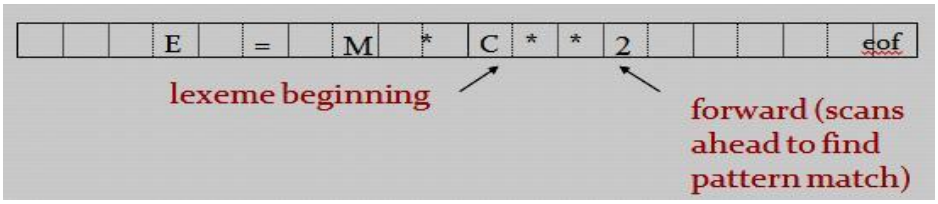
The three general approaches to the implementation of a lexical analyzer

1. Use a lexical analyzer generator such as the Lex compiler to produce a regular expression based specification. In this case, the generator provides for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems programming languages using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and reading of input.

The lexical analyzer is the only phase of the compiler that reads the source program character-by-character; it is possible to spend a considerable amount of time in the lexical analysis phase.

Buffer Pairs:

- Two pointers to the input buffer are maintained.
- The string of characters between the pointers is the current lexeme.
- Initially, both pointers point to the first character of the next lexeme to be found.
- Forward pointer, scans ahead until a match for a pattern is found.
- Once the next lexeme is determined, the forward pointer is set to the character at its right end.
- After the lexeme is processed, both pointers are set to the character immediately past the lexeme.
- The comments and white space can be treated as patterns that yield no token.
- A buffer into two N-character halves, where N is the no.of characters on one disk block, eg. 1024 or 4096.



- If the forward pointer is to move past the halfway mark, the right half is filled with N new input characters.

- If the forward pointer is to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

Code to advance forward pointer

if *forward* at the end of first half **then**

begin reload second half ;

forward : = forward + 1;

end

else if *forward* at end of second half **then begin**

reload first half ;

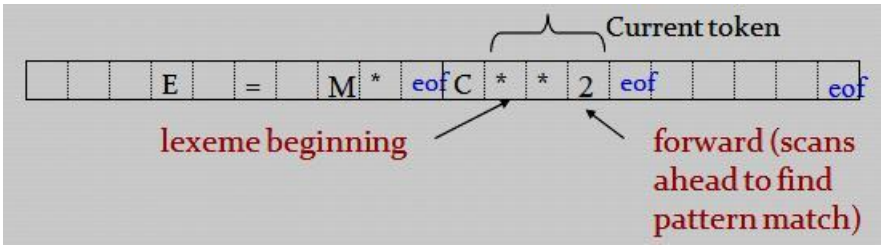
move *forward* to beginning of first half

end

else *forward : = forward + 1;*

Sentinels:

- The *sentinel* is a special character that cannot be part of the source program.
- Each buffer half to hold a sentinel character at the end (eof).



Lookahead code with

sentinels: *forward* : =

forward + 1 ;

if *forward* □ = **eof then begin**

if *forward* at end of first half **then begin**

reload second half ;

forward : = *forward* + 1

end

else if *forward* at end of second half **then begin**

reload first half ;

move *forward* to beginning of first half

end

else / * eof within buffer signifying end of input *****

/ terminate lexical analysis

end

9. Explain the specification of tokens? (11 marks)

Regular expressions are an important notation for specifying lexeme patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

(i) Strings and Languages:

- The term *alphabet or character class* denotes any finite set of symbols. Typical examples of symbols are letters and characters.
- The set {0, 1} is the *binary alphabet*.
- ASCII and EBCDIC are two examples of computer alphabet.
- A ***string*** over some alphabet is a finite sequence of symbols drawn from that alphabet.
- The length of string s, usually written |s|, is the number of occurrences of symbols in s.
- The *empty string* denoted Ɛ, is a special string of length zero.
- The term ***language*** denotes any set of strings over some fixed alphabet. Abstract languages like □, the empty set, or {□},the set containing only the empty string, are languages.
- If x and y are strings, then the *concatenation* of x and y is also string, denoted xy, is the string formed by appending y to x.
- For example, if x = ban and y = ana, then xy = banana.
- The empty string Ɛ is the identity element under concatenation; that is, for any string s, SƐ = ƐS= s.

(ii) Operations on Languages:

There are several important operations that can be applied to languages.

In lexical analysis

- Union
- Concatenation
- Closure

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
concatenation of L and M written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written L*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L* denotes "zero or more concatenations of " L
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of " L

Example:

- Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z } and D be the set of digits {0,1,.. .9}.
- L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits.

1. L U D is the set of letters and digits
2. LD is the set of strings consisting of a letter followed by digit
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including e, the empty string.
5. L (L U D)* is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

(iii) Regular Expressions:

- Regular expression is a notation for describing string. In Pascal, an identifier is a letter followed by zero or more letter or digits.
- The Pascal identifier as

letter (letter | digit) *

The rules is the specification of language denoted by

1. □ is a regular expression that denotes {□}, the set containing empty string.
2. If ***a*** is a symbol in □, then *a* is a regular expression that denotes {*a*}, the set containing the string *a*.
3. Suppose ***r*** and ***s*** are regular expressions denoting the language L(*r*) and L(*s*), then
   - **a)** (*r*) |(*s*) is a regular expression denoting L(*r*) □ L(*s*).
   - **b)** (*r*)(*s*) is regular expression denoting L (*r*) L(*s*).
   - **c)** (*r*) * is a regular expression denoting (L (*r*))*.
   - **d)** (*r*) is a regular expression denoting L (*r*).

A language denoted by a regular expression is said to be a *regular set.*

Unnecessary parentheses can be avoided in regular expression

1. The unary operator * has the highest precedence and is left associative.
2. Concatenation has the second highest precedence and is left associative.
3. | has the lowest precedence and is left associative.

(iv) Regular Definitions:

- For notation, give names to regular expressions and to define regular expressions using these names as if they were symbols.
- If □ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

d1□r

1

d2□r

2

. . .

dn□rn

where each di is a distinct name, and each ri is a regular expression.

Example:

1. The set of *Pascal identifier* is the set of strings of letters and digits beginning with a letter. The regular definition is

letter □ A | B | C | … | Z | a | b | …
| z **digit** □ 0 | 1 | 2 | … | 9
id □ **letter (letter | digit)***

2. *Unsigned numbers* in Pascal are strings such as 5280, 39.37, 6.336E4 or 1.894E-4.

The regular definition is

digit □ 0 | 1 | 2 | … | 9
digits □ **digit digit***

optional_fraction □ **. digits** | □
optional_exponent□ (**E** (+ | -| □) **digits**) |
□**num** □ **digits optional_fraction**
optional_exponent

(v) Notational Shorthands:

1. **One or more instances**

Unary postfix operator + means "one or more instances of".

2. **Zero or one instance**

Unary postfix operator ? means " zero or one instances of ".The regular definition for

num digit □ 0 | 1 | 2 | … | 9

digits □ **digit**$^{+}$ **optional_fraction** □ (. **digits**) ?

optional_exponent □ (E (+ | -) ? **digits)** ?

num □ **digits optional_fraction optional_exponent**

3. **Character classes**
   - The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
   - The character class such as [a-z] denotes the regular expression a | b | … | z.
   - Using character classes, we describe identifiers as being strings generated by the regular expression,

[A – Z a – z] [A – Z a – z 0 – 9] *

10. Illustrate the steps involved in the recognition of tokens? (11 marks) (NOV 2011)(MAY 2013)

We considered the problem of how to specify tokens and recognize them.

Consider the following

grammar *stmt* □ **if** *expr*

then *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| □

expr □ *term* **relop** *term*

| *term*

term □ *id*

| *num*

where the terminals **if, then**, **else**, **relop**, **id**, and **num** generate set of strings given by the following regular definitions:

if □ if **then**

□ then **else**

□ else

relop □ < | <= | > | >= | = | < >

id □ **letter (letter | digit)***

num □ **digit** + **(. digit** + **) ? (E(+ | -) ? digit** + **) ?**

The lexical analyzer will recognize the **keywords** *if, then, else,* as well as the lexemes denoted by ***relop, id***, and ***num.***

We assume lexemes are separated by **white space** consisting of *blanks, tabs, and newlines*. In lexical analyzer will strip out white space.

delim □ **blank** | **tab** | **newline**

ws □ **delim** +

If a match for ***ws*** is found, the lexical analyzer does not return a token to the parser.

- To construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value, using the translation table.
- The attribute values for the relational operators are given by the symbolic constants LT, LE, EQ, NE, GT,GE.

Regular Expression	Token	Attribute-Value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

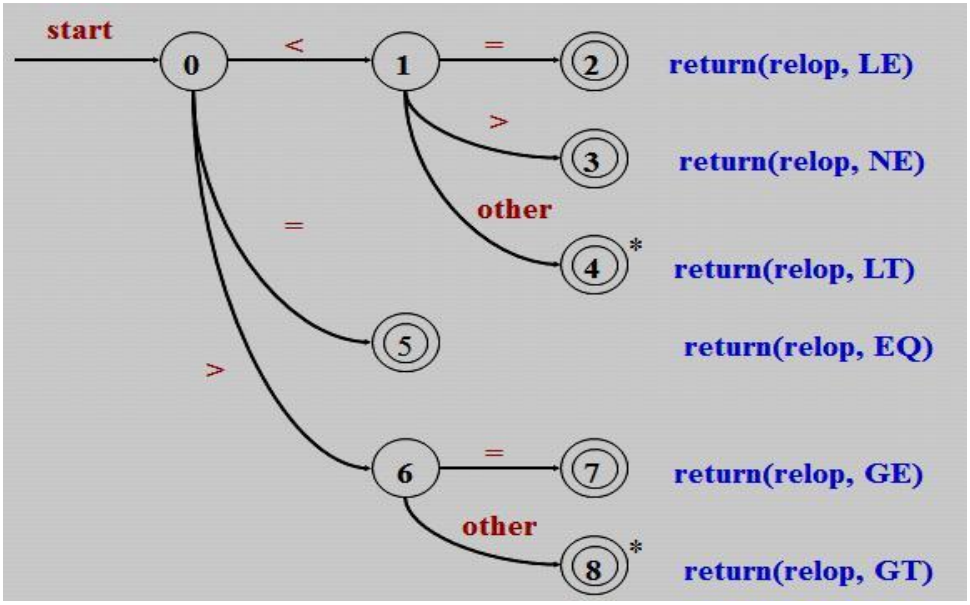
Regular expression pattern for tokens

Transition diagram:

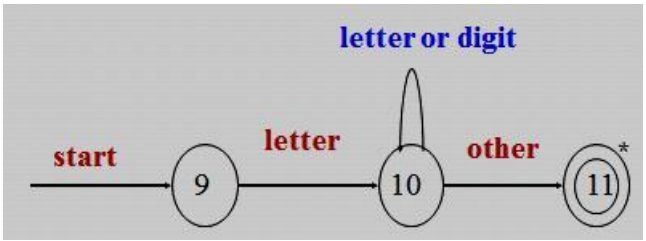
- An intermediate step in the construction of a lexical analyzer, produce a stylized flowchart called ***a transition diagram.***
- Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
- Transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
- Moving from position to position in the diagram as characters are read.
- Positions in a transition diagram are drawn as circles and are called **states.**
- The states are connected by arrow, called **edges**.
- Edges leaving state **s** have **labels** indicating the input characters that can next appear after the transition diagram has reached state ***s***.
- The **label other** refers to any character that is not indicated by any of the other edges leaving s.
- Transition diagram are **deterministic**, ie no symbol can match the labels of two edges leaving one state.

- One state is labeled as the ***start state***; it is the initial state of the transition diagram where control resides when we begin to recognize a token.
- Certain states may have actions that are executed when the flow of control reaches that state.
- On entering a state we read the next input character.
- If there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge.
- Otherwise we indicate failure.

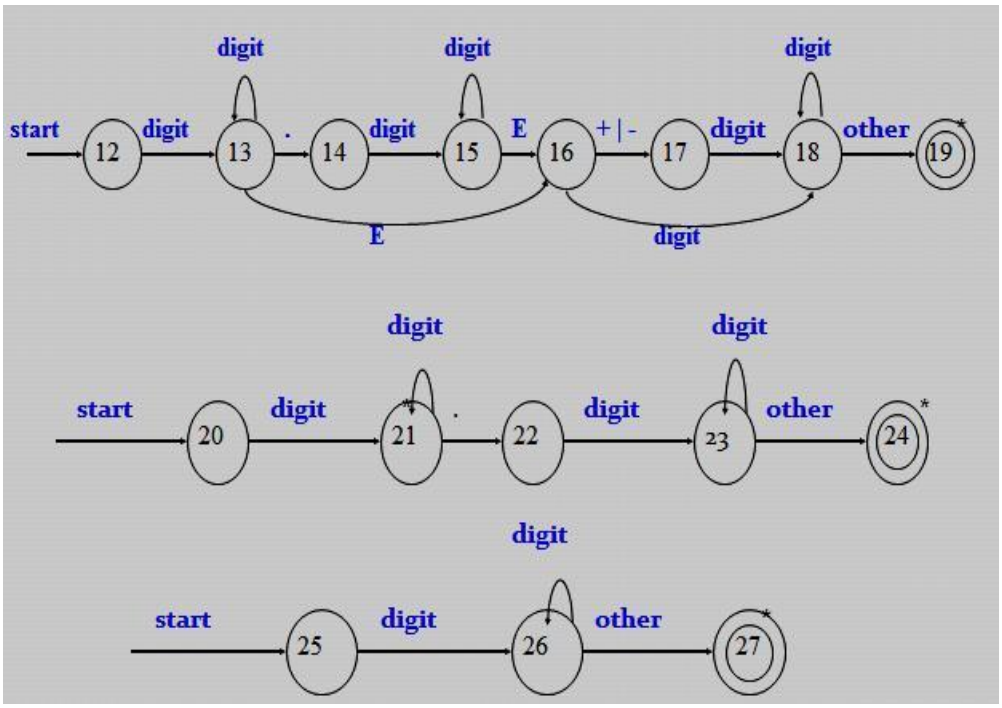
Transition diagram for relational operators:



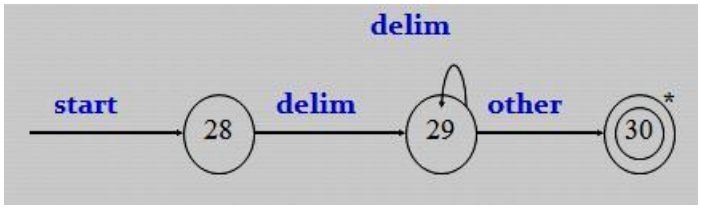
Transition diagram for identifiers and keywords:



Transition diagram for digits:



Transition diagram for delim:



Implementing a Transition diagram:

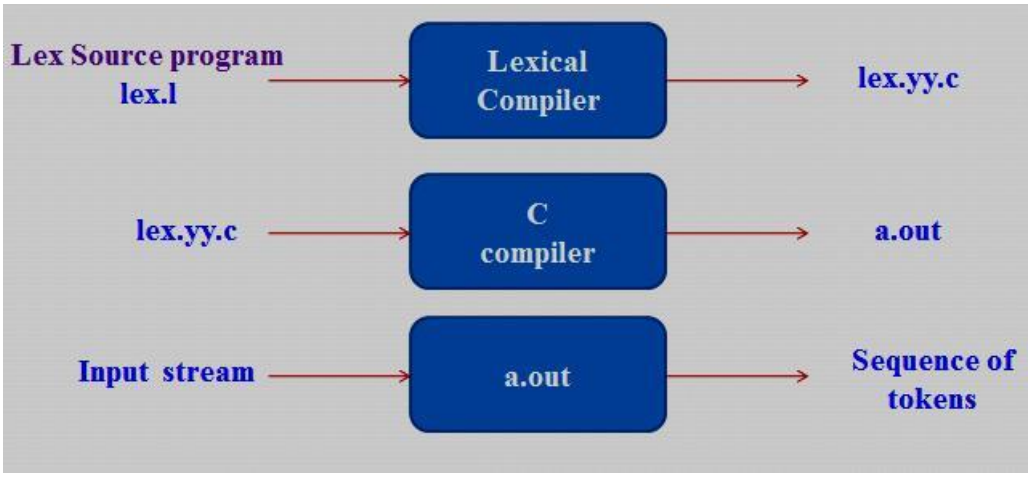
- A sequence of transition diagram can be converted into a program to look for the tokens by the diagrams.
- The systematic approach that works for all transition diagram and constructs programs whose size is proportional to the number of states and edges in the diagrams.

11. Elaborate on the language for specifying lexical analyzer. (6 marks) (NOV 2013)

- Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
- The use of regular expressions for specifying tokens patterns.
- A particular tool called ***Lex,*** which is used to specify lexical analyzer for a variety of languages.
- We refer to the tool as the ***Lex compiler*** and to its input specification as the Lex language.

Creating a lexical analyzer with Lex:

1. First, a specification of a lexical analyzer is prepared by creating a program ***lex.l*** in the Lex language.
2. Then, ***lex.l*** is run through the ***Lex compiler*** to produce a *C program* ***lex.yy.c.***
3. The program ***lex.yy.c*** consists of tabular representation of a transition diagram constructed from regular expression of ***lex.l***, together with a standard routine that uses the table to recognize lexemes.
4. The actions associated with regular expressions in *lex.l* are pieces of *C code* and are carried over directly to ***lex.yy.c.***
5. Finally ***lex.yy.c*** is run through the ***C compiler*** to produce an object program ***a.out***, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specifications:

A Lex program consists of three parts: ***declarations***

%%

translation rules

%%

auxiliary procedures

- The *declarations section* includes declarations of variables, manifest constants, and regular definitions.
- The *translation rules* of a Lex program are statement of the form

  *p*1 {

  action1} *p*2

  { *action2*}

  …… …..

  *p*n { *actionn*}

  where each ***pi*** is a regular expression and each ***actioni*** is a program fragment describing what action the lexical analyzer should take when pattern matches a lexeme.
- The *auxiliary procedures* are needed by the actions. These procedures can be compiled separately and loaded with the lexical analyzer.

A lexical analyzer created by Lex behaves with a parser in the following manner.

- When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions ***pi***.
- Then it executes ***actioni.***
- The lexical analyzer returns a single quantity, the token, to the parser.
- To pass an attribute value with the information about the lexeme, we can set a global variable called ***yylval.***

Lex Program for the tokens:

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */

delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
```

```
%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = install_id(); return(ID); }
{number}    {yylval = install_num(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP); }
"<="        {yylval = LE; return(RELOP); }
"="         {yylval = EQ; return(RELOP); }
"<>"        {yylval = NE; return(RELOP); }
">"         {yylval = GT; return(RELOP); }
">="        {yylval = GE; return(RELOP); }
%%

install_id()
{
        /* procedure to install the lexeme, whose first character is pointed to by
           yytext, and whose length is yyleng, into the symbol table and return a
           pointer */
}

install_num()
{
         /* similar procedure to install a lexeme that is a number */
}
```

IMPORTANT QUESTIONS

2 MARKS

1. What is hierarchical analysis? **(NOV 2011) (Ref.Qn.No.8, Pg.no.3)**
2. What is the major advantage of a lexical analyzer generator? **(NOV 2011) (Ref.Qn.No.30, Pg.no.8)**
3. List out the parts on Lex specifications. **(MAY 2012) (Ref.Qn.No.44, Pg.no.10)**
4. What is Compiler? **(MAY 2012) (NOV 2012) (Ref.Qn.No.1, Pg.no.2)**
5. What is transition diagram? **(NOV 2012) (Ref.Qn.No.41, Pg.no.9)**
6. Why separate lexical analysis phase is required? **(MAY 2013) (Ref.Qn.No.29, Pg.no.7)**
7. State the function of front end and back end of a compiler phase. **(MAY 2013) (Ref.Qn.No.22,23, Pg.no.6)**
8. State with example the cousins of compilers. **(NOV 2013) (Ref.Qn.No.17, Pg.no.5)**
9. List the role of lexical analyzer? **(NOV 2013) (Ref.Qn.No.27, Pg.no.7)**

11 MARKS

NOV 2011(REGULAR)

1. Draw the different phases of a compiler and explain. **(Ref.Qn.No.3, Pg.no.16)**

(OR)

2. How to recognize the tokens? **(Ref.Qn.No.10, Pg.no.34)**

MAY 2012(ARREAR)

1. Explain the Cousins of the compiler. **(Ref.Qn.No.4, Pg.no.22)**

(OR)

2. Discuss the Phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

NOV 2012(REGULAR)

1. Explain the phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

(OR)

2. Discuss the role of the lexical analyzer. **(Ref.Qn.No.7, Pg.no.27)**

MAY 2013(ARREAR)

1. a) State the different compiler construction tools and their use. (6) **(Ref.Qn.No.5, Pg.no.25)** b) Illustrate the steps involved in the recognition of tokens. (5) **(Ref.Qn.No.10, Pg.no.34)**

(OR)

2. With a neat sketch discuss the functionalities of various phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

NOV 2013 (REGULAR)

1. Describe the different stage of a compiler with an example. Consider an example for a simple arithmetic expression statement. **(Ref.Qn.No.3, Pg.no.16)**

(OR)

2. Explain the buffered I/O with sentinels. Elaborate on the language for specifying lexical analyzer. **(Ref.Qn.No.8, Pg.no.29) (Ref.Qn.No.11, Pg.no.38)**

UNIT IV

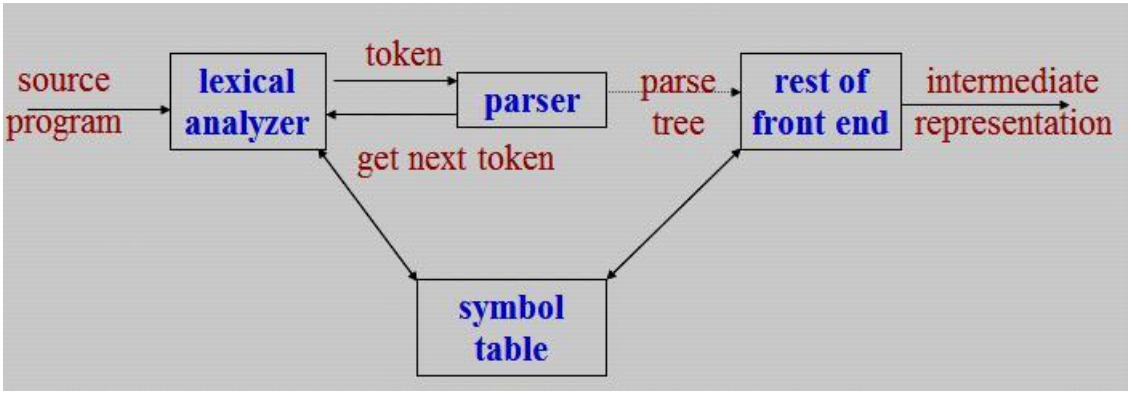
PARSING

Parsing: Role of Parser – Context free Grammars – Writing a Grammar – Predictive Parser – LR Parser. **Intermediate Code Generation:** Intermediate Languages – Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back Patching – Procedure Calls.

2 MARKS

1. What is the role of parser?

- In compiler model, parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source program.
- The parser should report any syntax errors in an intelligible fashion.



2. What is meant by 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. It obtains a string of tokens from the lexical analyzer, verifies that the string generated by the grammar for the source language.

3. What are the types of Parser?

There are three general types of parsers for grammars.

1. Universal parsing methods
   - Cocke-Younger –Kasami algorithm and
   - Earley's algorithm
2. Top down parser
3. Bottom up parser

4. What are the different levels of syntax error handler?

- Lexical, such as misspelling an identifier, keyword, or operator
- Syntactic, such as an arithmetic expression with unbalanced parentheses
- Semantic, such as an operator applied to an incompatible operand
- Logical, such as an infinitely recursive call

5. What are the goals of error handler in a parser?

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

6. What are error recovery strategies in parser?

- Panic mode
- Phrase level
- Error productions
- Global correction

7. Define context free grammar? (NOV 2011)

A Context Free Grammar (CFG) consists of terminals, non-terminals, a start symbol and productions.

The Grammar G can be represented as G = (V, T, S, P)

- V is a set of non-terminals
- T is a set of terminals
- S is a start symbol
- P is a set of production rules

Production rules are given in the following form

Non terminal $\rightarrow$ (V U T)*

8. Define derivation.

Derivation is the top-down construction of parse tree. The production treated as a rewriting rule in which the non-terminal on the left is replaced by the string on the right side of the production.

9. What is left-most and right-most derivation?

- The left-most non-terminal in each derivation step, this derivation is called as left-most derivation.
- The right-most non-terminal in each derivation step, this derivation is called as right-most derivation (Canonical derivation).

10. What is Parsing Tree? (MAY 2012)

- A parse tree can be viewed as a graphical representation for a derivation.
- The leaves of a parse tree are terminal symbols.
- Inner nodes of a parse tree are non-terminal symbols.

11. Define yield of the string?

The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right; they constitute a sentential form called the yield or frontier of the tree.

12. Define ambiguous. (MAY 2012)

- A grammar that produces more than one parse tree for a sentence is said to be *ambiguous.*

13. Define ambiguous grammar.

- An ambiguous grammar is one that produces more than one left most or more than one right most derivation for the same sentence.

14. What is left recursion?

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $\mathbf{A} \overset{+}{\Rightarrow} \mathbf{A}\alpha$**.**
- Top-down parsing techniques *cannot* handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

Example: The left recursive pair of productions $\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \boldsymbol{\beta}$ could be replaced by non- left- recursive productions

$\mathbf{A} \rightarrow \boldsymbol{\beta}\mathbf{A'}$

$\mathbf{A'} \rightarrow \alpha\,\mathbf{A'} \mid \boldsymbol{\varepsilon}$

15. Define left factoring?

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

16. What are the problems with top down parsing?

The following are the problems associated with top down parsing:

- Backtracking
- Left recursion
- Left factoring
- Ambiguity

17. Define top down parsing?

- Top-down parser viewed as an attempt to find the left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- Top down parsing called recursive descent that may involve backtracking ie. making repeated scanning of the input.

18. What is meant by recursive-descent parser?

- A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser.
- This recursive-descent parser called predictive parsing.

19. Briefly describe the LL (k) items. . (NOV 2013)

In LL (k) the first "L " scanning the input from left to right and

second "L" producing a leftmost derivation and

the "1" one input symbol of lookahead at each step

20. What are the possibilities of non-recursive predictive parsing?

a) If X = a = $, the parser halts and announces successful completion of parsing.

b) If X = a = $, the parser pops X off the stack and advances the input pointer to the next symbol.

c) If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.

21. Write the algorithm for FIRST and FOLLOW.

FIRST

1. If X is terminal, and then FIRST(X) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is non-terminal and $X \rightarrow Y_1,Y_2....Y_k$ is a production, then place a in FIRST(X) if for some i , a is in FIRST(Y_i) , and ε is in all of FIRST(Y_1),...FIRST(Y_{i-1});

FOLLOW

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST (β) except for ε is placed in FOLLOW (B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

22. What is bottom up parser?

- Bottom-up parsing is also known as shift-reduce parsing.
- Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

23. Define handle?

- A **handle** of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

24. What is meant by handle pruning?

- A rightmost derivation in reverse can be obtained by *handle pruning*.
- If w is a sentence of the grammar at hand, then w = γ_n, where γ_n is the *n*th right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 => \gamma_1 ... => \gamma_{n-1} => \gamma_n = w$$

25. What is meant by viable prefixes?

- The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes.*
- An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential

26. Define LR (k) parser?

LR parsers can be used to parse a large class of context free grammars. The technique is called ***LR (K)*** parsing.

- "L" is for left-to-right scanning of the input
- "R" for constructing a right most derivation in reverse
- "k" for the number of input symbols of lookahead that are used in making parsing decisions.

27. Mention the types of LR parser?

The three methods in LR parser

- Simple LR (SLR) parser
- Canonical LR (CLR) parser
- Lookahead LR (LALR) parser

28. What are the techniques for producing LR parsing Table?

1. Shift s, where s is a state
2. Reduce by a grammar production $A \rightarrow \beta$
3. Accept and
4. Error

29. What are the two functions of LR parsing algorithm?

The two functions in LR parsing algorithm are

- Action function
- GOTO function

30. Define LALR grammar?

The Lookahead (LALR) parser method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming language can be expressed conveniently by an LALR grammar. If there are no parsing action conflicts, then the given grammar is said to be an LALR (1) grammar. The collection of sets of items constructed is called LALR (1) collections.

31. Define SLR parser?

The parsing table consisting of the parsing action and goto function determined by constructing an SLR parsing table algorithm is called SLR(1) table. An LR parser using the SLR (1) table is called SLR (1) parser. A grammar having an SLR (1) parsing table is called SLR (1) grammar.

32. Differentiate phase and pass. . (NOV 2012)

Phase	**Pass**
▪ Phase is often used to call such a single independent part of a compiler. ▪ It is used in complier. ▪ It has lexical, syntax, semantic analyzer, intermediate code generator, code optimizer, and code generator.	▪ Number of passes of a compiler is the number of times it goes over the source. ▪ It also used in compiler. ▪ Compilers are indentified as one-pass or multi-pass compilers. ▪ I t is easier to write a one-pass compiler and also they perform faster than multi-pass compilers.

33. State the function of an intermediate code generator. (MAY 2013)

A source program can be translated directly into target language, some benefits of using machine-independent intermediate form:

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

34. What are the syntax-directed methods?

The syntax-directed methods can be used to translate into intermediate form programming language constructs such as

- Declaration
- Assignment statements
- Boolean Expression
- Flow of control statements

35. What are the different forms of Intermediate representations? (NOV 2013)

The three kinds of intermediate representations are

i. Syntax trees

ii. Postfix notation

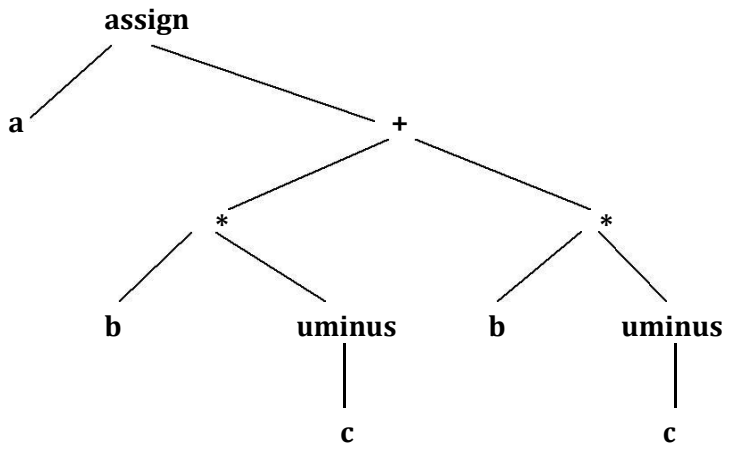
iii. Three - address code

36. How can you generate three-address code?

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

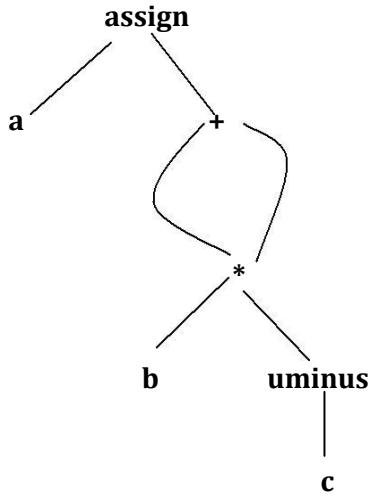
37. What is a syntax tree? Draw the syntax tree for the assignment statement.

- A syntax tree depicts the natural hierarchical structure of a source program.
- The syntax tree for the assignment statement **a: = b * -c + b * -c.**



38. Draw the dag for the assignment statement: a: = b * -c + b * -c. (NOV 2011)

- Directed Acyclic Graph (DAG) gives more compact way for common sub-expressions are identified.
- The DAG for the assignment statement **a: = b * -c + b * -c.**



39. Define postfix notation?

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

  a b c uminus * b c uminus * + assign

40. What are the functions used to create the nodes of syntax trees?

- mkunode(op, child)
- mknode(op, left, right)
- mkleaf(id, entry)

41. Define three-address code. (NOV 2012)

- 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 fixed or floating-point arithmetic operator, or a logical operator on boolean-valued data.

42. Construct three address codes for the following a: = b * -c + b * -c.

The three address code

as 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

43. List the types of three address statements.

The types of three address statements are

1. Assignment statements
2. Assignment Instructions
3. Copy statements
4. Unconditional Jumps
5. Conditional jumps
6. Procedure calls and return
7. Indexed assignments
8. Address and pointer assignments

44. What are the various implementing three-address statements?

The three implementation of three address statements are

i. Quadruples

ii. Triples

iii. Indirect triples

45. What is a quadruple?

- A *quadruple* is a record structure with four fields, such as

  op, argl, arg2, and ***result***

- The *op* field contains an internal code for the operator.

- The three-address statement x := y op z is represented by placing y in arg 1, z in arg 2, and x in result.

46. What are triples?

- The Three-address statements can be represented by records with only three fields:

  op, arg1 and ***arg2***

- The fields *arg l* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure. Since three fields are used, this intermediate code format is known as ***triples.***

- This method is used to avoid temporary names into the symbol table.

47. Define indirect triples. Give the advantage?

- Listing pointers to triples rather than listing the triples themselves. This implementation is called ***indirect triples.***

Advantages:

- It can save some space compared with quadruples, if the same temporary value is used more than once.

48. Write a short note on declarations?

- Declarations in a procedure, for each local name, we create a symbol table entry with information like the type and the relative address of the storage for the name.

- The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

- The procedure ***enter (name, type, offset)*** create a symbol table entry for *name,* its *type* and relative address *offset* in its data area.

49. What are the semantic rules are defined in the declarations operations?

The semantic rules are defined by the following ways

1. mktable(previous)
2. enter(table, name, type, offset)
3. addwidth(table, width)
4. enterproc(table, name, newtable)

50. What are the two primary purposes of Boolean Expressions?

In Boolean expressions have two primary purposes

1. They are used to compute logical values
2. 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.

51. Define Boolean Expression.

- Boolean expressions which are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions.
- Relational expression of the form **E1 relop E2,** where E1 and E2 arithmetic expressions.
- Consider Boolean Expressions with the following grammar:

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

52. What are the methods of translating Boolean expressions?

There are two principal methods of representing the value of a Boolean expression.

1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.
2. The second principal method of implementing boolean expression is by flow of control that is representing the value of a Boolean expression by a position reached in a program.

53. What are the three address code for a *or b and not c* ?

The three address sequence for a or b and not c t_1

:= not c

t_2 := b and t_1

t_3 := a or t_2

54. What is meant by Shot-Circuit or jumping code?

Translate a Boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short-circuit” or “jumping” code.

55. Write a three address code for the expression a<b or c<d and e<f?

The three address code as

100: ***if*** a< b goto ***103***

101: t_1 := 0

102: goto ***104***

103: t_1 := 1

104: ***if*** c< d goto ***107***

105: t_2 := 0

106: goto ***108***

107: t_2 := 1

108: ***if*** e< f goto ***111***

109: t_3 := 0

110: goto ***112***

111: t_3 := 1

112: t_4 := t_2 ***and*** t_3

113: t_5 := t_1 ***or*** t_4

56. Define back patching.

- Backpatching can be used to generate 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.
- 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. This subsequent filling of addresses for the determined labels is called

  Backpatching.

57. What are the three functions of backpatching?

The three functions in backpatching are

1. makelist(i) – create a new list.
2. merge(p_1,p_2) – concatenates the lists pointed to by p1 and p2.
3. backpatch(p,i) – insert i as the target label for the statements pointed to by p.

58. Derive the first and follow for the follow for the following grammar. (MAY 2013)

S → 0|1|AS0|BS0 A → **ε** B → **ε**

Computation for FIRST:

FIRST(S) = {0, 1} U FIRST (A) U FIRST (B) = {0, 1} U {ε} U {ε} = **{0, 1, ε}**

FIRST (A) = **{ε}**

FIRST (B) = **{ε}**

Computation for FOLLOW:

FOLLOW (S) = {$} U {0} U {0} = **{$, 0}**

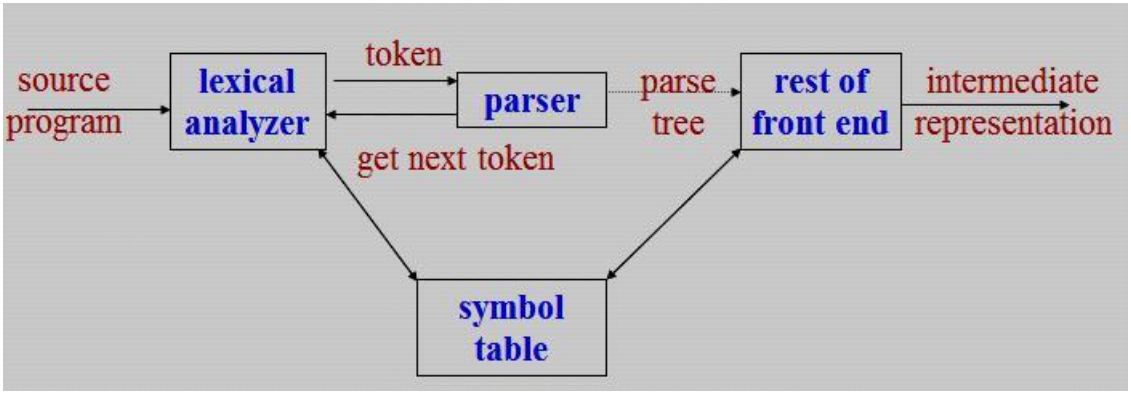
FOLLOW (A) = FOLLOW(S) = **{$, 0}**

FOLLOW (B) = FOLLOW(S) = **{$, 0}**

11 MARKS

1. Explain the role of the parser? (11 marks) (MAY 2012)

- In compiler model, parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source program.
- The parser should report any syntax errors in an intelligible fashion.
- It should also recover from commonly occurring errors so it can continue processing the remainder if it's input.



Position of parser in compiler model

There are three general types of parsers for grammars.

1. Universal parsing methods → too inefficient to use in production compilers.
   - Cocke-Younger –Kasami algorithm and
   - Earley's algorithm
2. Top down parser → it builds parse trees from the top (root) to the bottom (leaves).
3. Bottom up parser → it start from the leaves and work up to the root.

- In both cases, the input to the parser is scanned from left to right, one symbol at a time.
- The most efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Syntax error handling:

- If a compiler to process only correct programs, its design and implementation would be greatly simplified.

The program can contain errors at many different levels of syntax error handler

- *Lexical,* such as misspelling an identifier, keyword, or operator
- *Syntactic,* such as an arithmetic expression with unbalanced parentheses
- *Semantic,* such as an operator applied to an incompatible operand
- *Logical,* such as an infinitely recursive call

The error handler in a parser has simple to state goals:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Several parsing methods such as LL and LR methods, detect an error as soon as possible.

Error - Recovery Strategies:

There are many different strategies that a parser can recover from a syntactic error.

- Panic mode
- Phrase level
- Error productions
- Global correction

Panic mode recovery:

- This is the simplest method to implement and can be used by most parsing methods.
- On discovering an error, parser discards input symbols one at a time until one of the designated set of **synchronizing tokens** is found.
- The synchronizing tokens are usually delimiters such as semicolon or ***end.***
- It skips many inputs without checking additional errors, so it has an advantage of simplicity.
- It guaranteed not to go in to an infinite loop.

Phrase - level recovery

- On discovering an error, parser perform local correction on the remaining input;
- It may replace a prefix of the remaining input by some string that allows the parser to continue.
- Local correction would be to *replace a comma by a semicolon, delete an extra semicolon, or insert a missing semicolon.*

Error productions

- Augment the grammar with productions that generate the erroneous constructs.
- The grammar augmented by these error productions to construct a parser.
- If an error production is used by the parser, generate error diagnostics to indicate the erroneous construct recognized the input

Global correction

- Algorithms are used for choosing a minimal sequence of changes to obtain a globally least cost correction.
- Given an incorrect input string *x* and grammar G, these algorithms will find a parse tree for a related string y; such that the number of *insertions, deletions and changes of tokens* required to transform x into y is as small as possible.
- This technique is most costly in terms of time and space

2. Explain the Context Free Grammar (CFG)? (6 marks)

A ***Context Free Grammar (CFG)*** consists of terminals, non-terminals, a start symbol and productions.

The (CFG) Grammar G can be represented as G = (V, T, S, P)

- A finite set of terminals (The set of tokens)
- A finite set of non-terminals (syntactic-variables)
- A start symbol (one of the non-terminal symbol)
- A finite set of productions rules in the following form
  - $A \rightarrow \alpha$, where **A** is a non-terminal and α is a string of terminals and non-terminals including the empty string).
  - Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example:

The grammar with the following productions defines simple arithmetic

expressions. expr → expr op expr
expr → (expr)
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑

- In this grammar, the terminals symbols are

  id + - * / ↑ ( )
- The non-terminal symbols are ***expr*** and ***op***
- ***expr*** is the start symbol.

Notational Conventions:

1. These symbols are terminals:
   - **i)** Lower case letters in the alphabet such as a, b, c.
   - **ii)** Operator symbols such as +,-, etc.
   - **iii)** Punctuation symbols such as parenthesis, comma, etc.
   - **iv)** The digits 0, 1,……., 9.
   - **v)** Boldface strings such as ***id*** or ***if.***
2. These symbols are non-terminals
   - **i)** Upper case letters in the alphabet such as A, B, C.
   - **ii)** The letter S, when it appears, is usually the start symbol.
   - **iii)** Lower-case italic names such as ***expr*** or ***stmt***.
3. Upper-case letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*, that is, either non-terminals or terminals.
4. Lower-case letters late in the alphabet u, v, ..., z, represent strings of terminals.
5. Lower-case Greek letters α **,β**, γ represent strings of grammar symbols.
6. If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, $A \rightarrow \alpha_3$, ……, $A \rightarrow \alpha_k$ are all productions with, A on the left (A-productions), write as $A \rightarrow \alpha_1|\alpha_2|\alpha_3|...|\alpha_k$ the alternatives for A.
7. The left side of the first production is the start symbol.

Derivations:

- Derivational view gives a precise description of the top-down construction of a parse tree.
- The central idea is that a production is treated as a rewriting rule in which the non-terminals on the left is replaced by the string on the right side of the production.
- For example, consider the following grammar for arithmetic expressions, with the non-terminals E representing an expression.

  $$\mathbf{E \rightarrow E + E \mid E - E \mid E * E \mid (E) \mid - E \mid id}$$
- The production $E \rightarrow - E$ signifies that an expression preceded by a minus sign is also an expression. This production can be used to generate more complex expressions from simpler expressions by allowing us to replace any instance of an E by - E.

  $$\mathbf{E \Rightarrow -E}$$

- Given a grammar G with starts symbol, use the $\rightarrow^+$ relation to define L (G), the language generated by G.
- Strings in L (G) may contain only terminal symbols of G.
- A string of terminals w is in L (G) if and only if $S \rightarrow^+ w$. The string w is called a *sentence of G.*
- A language that can be, generated by a grammar is said to be a ***context-free language.***
- If two grammars generate, the same language, the grammars are said to be *equivalent*.

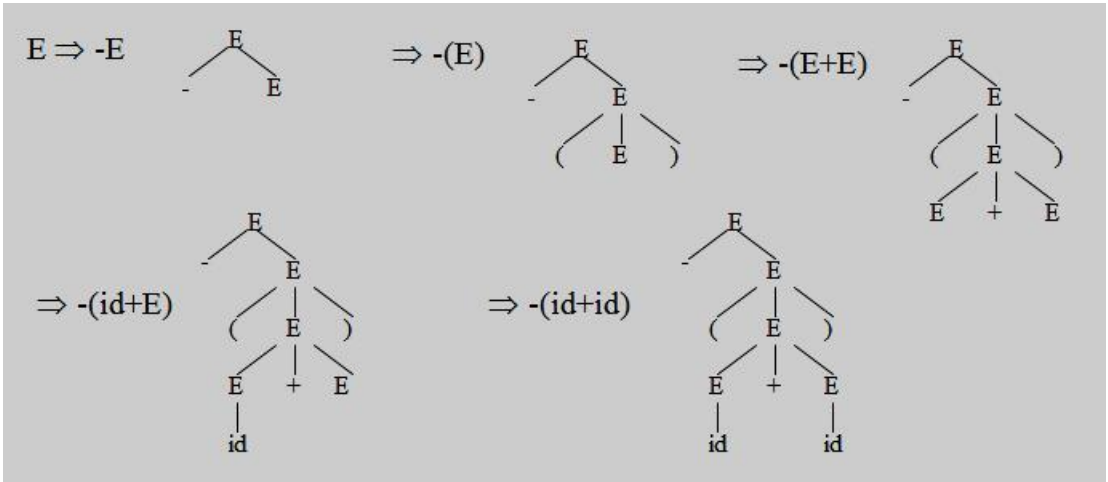
The string – (id+id) is a sentence of the grammar, and then the derivation is

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$

- At each derivation step, we can choose any of the non-terminals in the sentential form of G for the ***replacement.***
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as ***left-most derivation***.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as ***right-most derivation (Canonical derivation).***

Parse Trees and Derivations:

- A ***parse tree*** may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.
- Each interior node of a parse tree is labeled by some non-terminals A, and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this A was replaced in the derivation.
- The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right; they constitute a sentential form, called the ***yield or frontier of the tree.***



Ambiguity

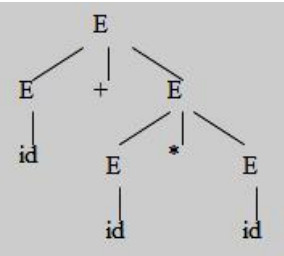
- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous***.
- An ***ambiguous grammar*** is one that produces more than one left most or more than one right most derivation for the same sentence.
- For the most parsers, the grammar must be unambiguous.
- The ***unambiguous grammar*** unique selection of the parse tree for a sentence.

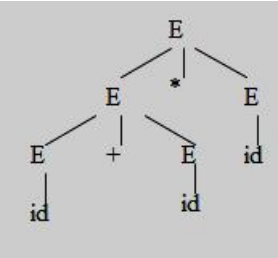
The sentence **id+id*id** has the two distinct *leftmost derivations:*

$E \Rightarrow E + E$
$\Rightarrow$ id + E
$\Rightarrow$ id + E * E
$\Rightarrow$ id + id * E
$\Rightarrow$ id + id * id

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow$ id + E * E
$\Rightarrow$ id + id * E
$\Rightarrow$ id + id * id

with the two corresponding *parse trees* are





3. Write the steps in writing a grammar for a programming language. (5 marks)(NOV 2013)

Grammars are capable of describing the syntax of the programming languages.

Regular Expressions vs. Context-Free Grammars:

- Every constructs that can be described by a regular expression can also be described by a grammar.
- For example the regular expression **(a | b)* abb,** the grammar is:

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \varepsilon$

which describe the same language, the set of strings of a's and b's ending in abb.

- Mathematically, the NFA is converted into a grammar that generates the same language as recognized by the NFA.

There are several reasons the regular expressions differ from CFG.

1. The lexical rules of a language are frequently quite simple. No need of any notation as powerful as grammars.
2. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.
3. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.
4. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

- Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords etc…
- Grammars are most useful in describing nested structures such as balanced parenthesis, matching begin - end's, corresponding if-then-else's and so on.

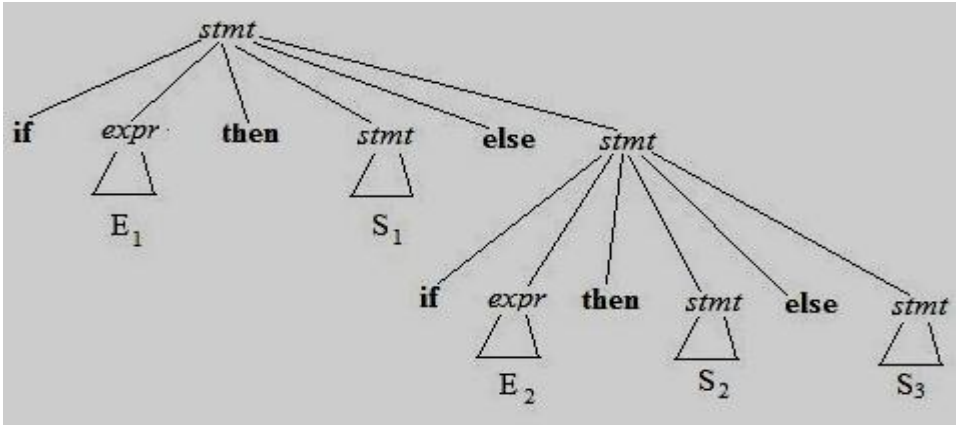
Eliminating Ambiguity:

- An ambiguous grammar can be rewritten to eliminate the ambiguity.
- Example for eliminate the ambiguity from the following "***dangling-else"*** → ***grammar:*** stmt **if** expr **then** stmt
  | **if** expr **then** stmt **else** stmt
  | **other**

Here **"other"** stands for any other statements. According to this grammar, the compound conditional statement

if E_1 ***then*** S_1 ***else if*** E_2 ***then*** S_2 ***else*** S_3

has the parse tree as

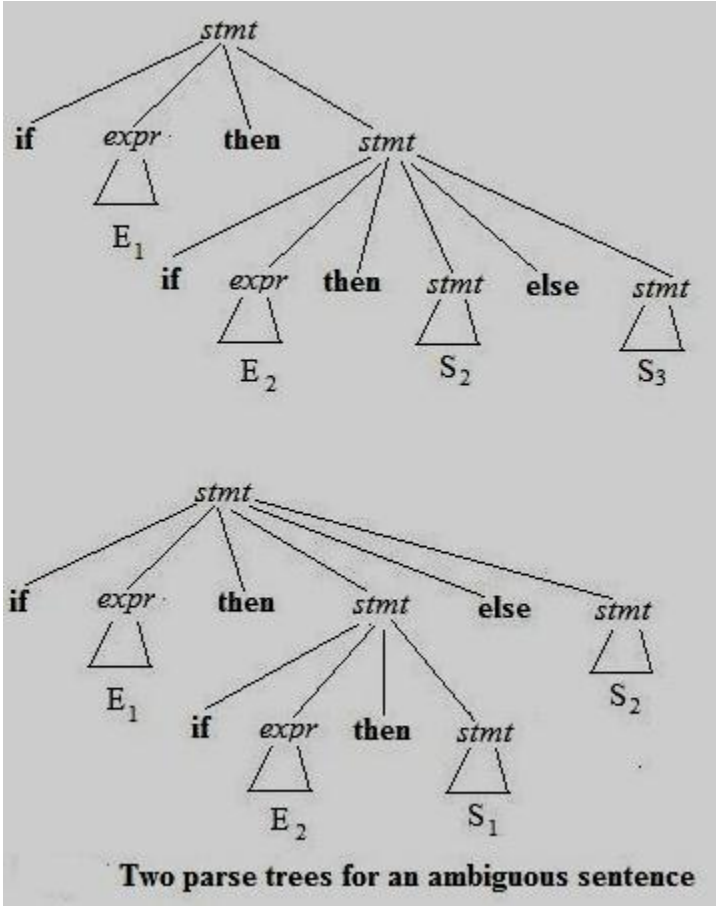


Parse tree for conditional statement

Grammar is ambiguous since the string

if E_1 ***then if*** E_2 ***then*** S_1 ***else*** S_2 has the

two parse trees



Two parse trees for an ambiguous sentence

- In all programming languages with conditional statements of this form, the first parse tree is preferred.
- The general rule is, "Match each ***else*** with the closest previous unmatched ***then***" this disambiguating rule can be incorporated directly into the

grammar. The unambiguous grammar will be:

stmt → matched_stmt

| unmatched_stmt

matched_stmt → ***if*** expr ***then*** matched_stmt ***else*** matchedstmt |

other

unmatched_stmt → ***if*** expr ***then*** stmt

| ***if*** expr t***hen*** matched_stmt ***else*** unmatched_stmt

Elimination of Left Recursion:

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation $\mathbf{A} \Rightarrow \mathbf{A}\alpha$.
- Top-down parsing techniques *cannot* handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- The left recursive pair of productions $\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \boldsymbol{\beta}$ could be replaced by non- left- recursive productions

$\mathbf{A} \rightarrow \boldsymbol{\beta}\mathbf{A'}$

$\mathbf{A'} \rightarrow \alpha\,\mathbf{A'} \mid \boldsymbol{\varepsilon}$

Algorithm to eliminating left recursion from a grammar:

Input: Grammar G with no cycles or ε-productions.

Output: An equivalent grammar with no left recursion.

Method: Note that the resulting non-left-recursive grammar may have ε-productions.

1. Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$
2. ***for*** i := 1 ***to*** n ***do begin***

    for j := 1 ***to*** i-1 ***do begin***

        replace each production of the form $A_i \rightarrow A_j\gamma$

        by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current Aj-productions;

    end

    eliminate the immediate left recursion among the A_i-productions

end

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, then rewrite the A-productions to defer the decision until the input to make the right choice.

In general, productions are of the form $\mathbf{A \rightarrow \alpha\beta_1 \mid \alpha\beta_2}$ then it is left factored as:

$\mathbf{A \rightarrow \alpha A'}$

$\mathbf{A' \rightarrow \beta_1 \mid \beta_2}$

Algorithm for Left factoring a grammar

Input: Grammar G.

Output: An equivalent left-factored grammar.

Method: For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α ≠ ε, i.e., there is a nontrivial common prefix, replace all the A productions $\mathbf{A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma}$ where γ represents all alternatives that do not begin with α by,

$$\mathbf{A \rightarrow \alpha A' \mid \gamma}$$

$$\mathbf{A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$$

4. Briefly write on Parsing techniques. Explain with illustration the designing of a Predictive Parser. (11 marks) (NOV 2013)

- The top-down parsing and show how to construct an efficient non-backtracking form of top-down parser called ***predictive parser.***
- Define the class ***LL (1) grammars*** from which predictive parsers can be constructed automatically.

Recursive-Descent Parsing:

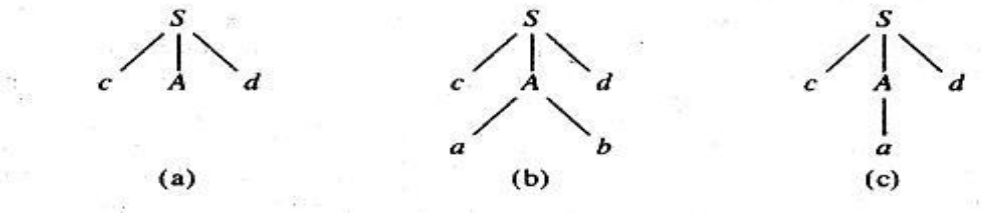
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- It is to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- The special case of recursive-descent parsing called predictive parsing, where no backtracking is required.
- The general form of top-down parsing, called recursive-descent, that may involve backtracking, ie, making repeated scans of input.
- However, backtracking parsers are not seen frequently.
- One reason is that backtracking is rarely needed to parse programming language constructs.
- In natural language parsing, backtracking is still not very efficient and tabular methods such as the dynamic programming algorithm.

Consider the grammar

$$\mathbf{S \rightarrow cAd}$$

$$\mathbf{A \rightarrow ab \mid a}$$

An input string ***w=cad,*** steps in top-down parse are as:



- A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

Predictive Parsers:

- For writing a grammar, eliminating left recursion from it and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parsing that needs no backtracking

  i.e., a ***predictive parser.***

- Flow-of-control constructs in most programming languages, with their distinguishing keywords, are usually detectable in this way.

- For example, if we have the productions

  stmt → **if** *expr* **then** *stmt* **else** *stmt*

  | while expr do *stmt*

  | begin *stmt_list* **end**

  then the keywords ***if, while,*** and ***begin*** that could possibly succeed to find a statement.

Transition Diagrams for Predictive Parsers:

Several differences between the transition diagrams for a lexical analyzer and a predictive parser.

- In case of parser, there is one diagram for each non-terminal.
- The labels of edges are tokens and non-terminals.
- A transition on a token means that transition if that token is the next input symbol.
- A transition on a non-terminal A is a call of the procedure for A.

To construct the transition diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar, and then left factor the grammar.

Then for each non-terminal A do the following:

1. Create an initial and final (return) state.
2. For each production A → X_1, X2 ... X_n, create a path from the initial to the final state, with edges labeled X_1, X_2,….,X_n.

Predictive Parser working:

- It begins in the start state for the start symbol.
- If after some actions it is in state **s** with an edge labeled by terminal **a** to state **t**, and if the next input symbol is **a**, then the parser moves the input cursor one position right and goes to state **t**.
- If, on the other hand, the edge is labeled by a non-terminal A, the parser instead goes to the start state for A, without moving the input cursor.
- If it ever reaches the final state for **A**, it immediately goes to state **t**, in effect having read **A** from the input during the time it moved from state **s** to **t**.

- Finally, if there is an edge from **s** to **t** labeled **ε**, then from state **s** the parser immediately goes to state **t**, without advancing the input.

- A predictive parsing program based on a transition diagrams attempts to match terminal symbols against the input and makes a potentially recursive procedure call whenever it has to follow an edge labeled by a non-terminal.

- A non-recursive implementation can be obtained by stacking the states **s** when there is a transition on non-terminal out of **s** and popping the stack when the final state for a non-terminal is reached.

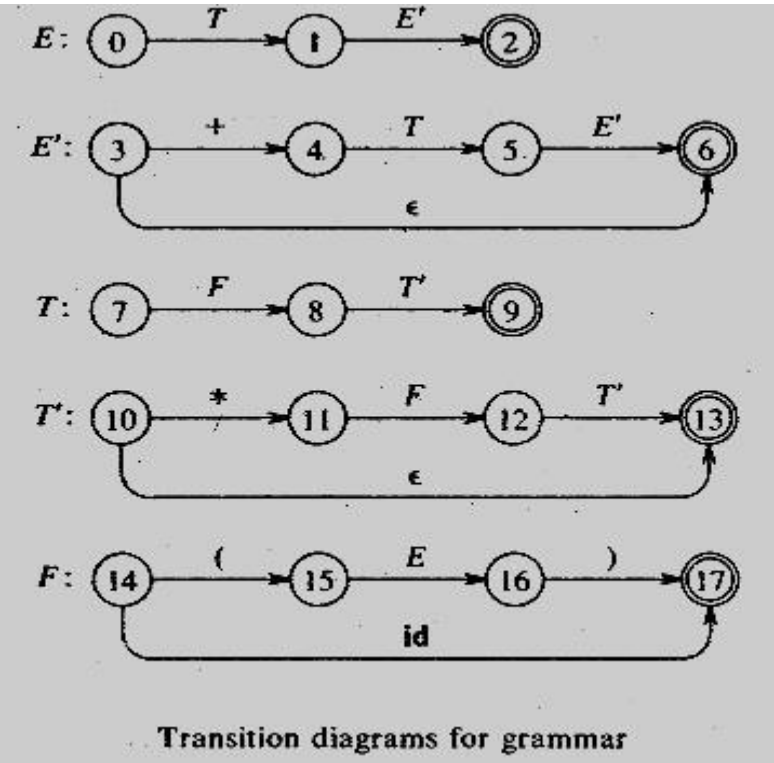
- Transition diagrams can be simplified by substituting diagrams in one another; these substitutions are similar to the transformations on grammars.

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Consider the following grammar for arithmetic expressions,

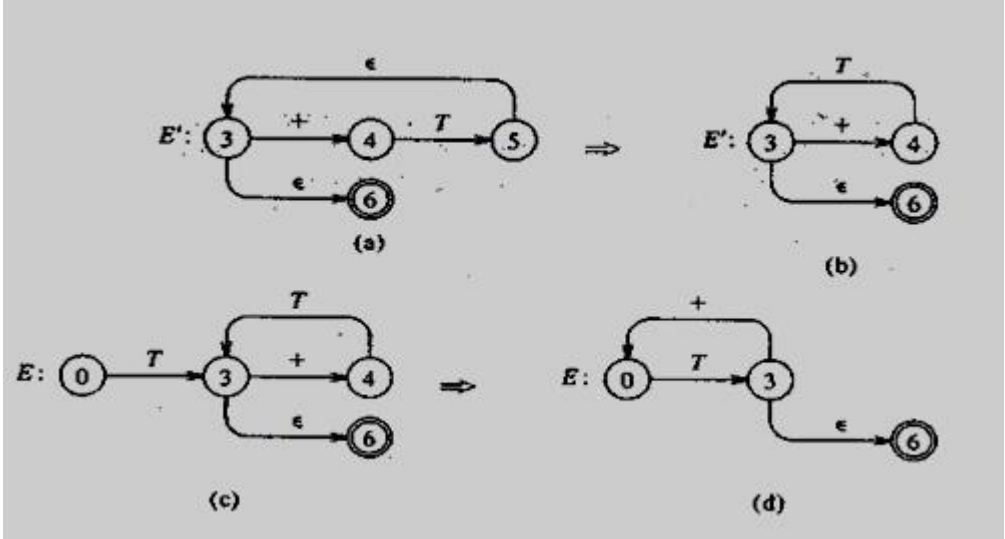
$$E \rightarrow T$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

Transition diagrams for grammar:

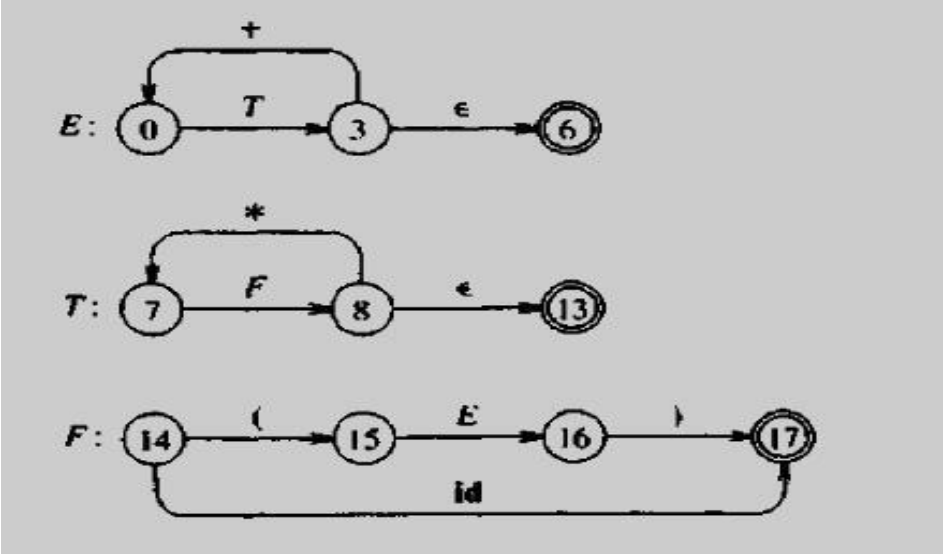


Transition diagrams for grammar

Simplified transition diagram:

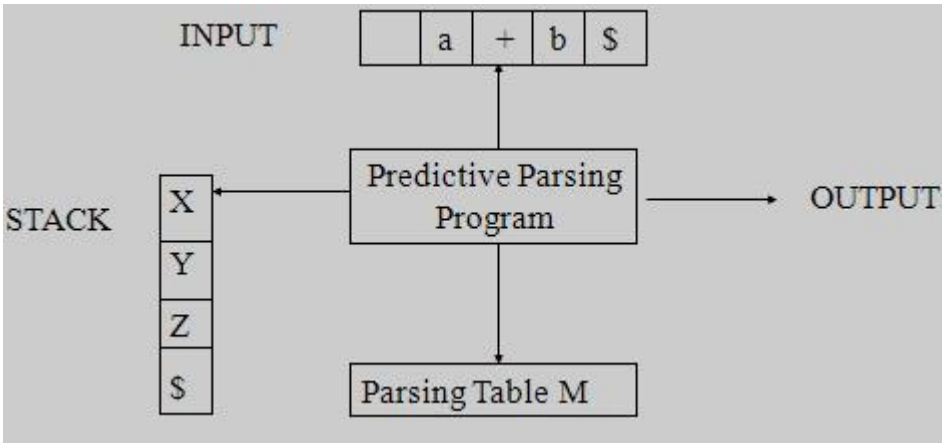


Simplified transition diagrams for arithmetic expressions:



5. Explain the Non-recursive predictive parsing? (11 marks)

A non recursive predictive parser by maintaining a stack explicitly, rather than implicitly through recursive calls. The key problem during predictive parser is that of determining the production to be applied for a non-terminal.



Model of a non-recursive predictive parser

A table-driven predictive parser has

- an input buffer,
- a stack,
- a parsing table; and
- an output stream.

- The ***input buffer*** contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
- The ***stack contains*** a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $.
- The ***parsing table*** is a two dimensional array M [A, a], where ***A*** is a non-terminal, and ***a*** is a terminal or the symbol $.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

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 a non-terminal, the program consults entry M [X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. For example M [X, a] = {X $\rightarrow$ UVW}, the parser replaces X on top of the stack by WVU (U on top).
4. If M[X, a] = error, the parser calls an error recovery routine.

Algorithm Non recursive predictive parsing:

Input: A string w and a parsing table M for grammar G.

Output: If w is in L (G), a leftmost derivation of w; otherwise an error indication.

Method: Initially, the parser is in a configuration in which it has $S on the stack with S, the start symbol of G on top; and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

```
set ip to point to the first symbol of
    w$; repeat
        let X be the top of the stack and a the symbol pointed by
        ip if X is a terminal or $ then
            if X=a then
            pop X from the stack and advance
            ip else error( )
        else  /* X is a non-terminal */
            if M [X ,a] = X → Y1 Y2 …YK then
                begin pop X from the stack;
                push YK … …Y2 Y1 on to the stack ,with Y1 on
                top; output the production X → Y1 Y2 …YK
                end
                else error( )
    until X= $ /* stack is empty */
```

ALGORITHM FOR FIRST:

1. If X is terminal, and then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → $Y_1, Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i , a is in FIRST(Y_i) , and ε is in all of FIRST(Y_1),…FIRST(Y_{i-1});

ALGORITHM FOR FOLLOW:

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production A → αBβ, then everything in FIRST (β) except for ε is placed in FOLLOW (B).
3. If there is a production A → αB, or a production A → αBβ where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

6. Construct the predictive parser for the following

grammar E → E + T | T

T → T * F | F

F → (E) | id

Compute FIRST and FOLLOW and also find the parsing table. The input string is id+id * id.

Solution:

The given grammar is

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

1. ELIMINATING LEFT RECURSION FROM THE GRAMMAR:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon F$

$\rightarrow (E) \mid id$

2. COMPUTATION OF FIRST:

FIRST (E) = FIRST (T) = FIRST (F) = { **(, id** }

FIRST (E`) = { **+, ε** }

FIRST (T) = FIRST (F) = { **(, id** }

FIRST (T`) = { ***, ε** }

FIRST (F) = { **(, id** }

3. COMPUTATION OF FOLLOW:

FOLLOW (E) = {**$**} **U** FOLLOW (E) = { **), $** }

FOLLOW (E') = FOLLOW (E) = { **), $** }

FOLLOW (T) = FOLLOW (E') **U** FIRST (E') = {), $} **U** {+} = { **+,), $** }

FOLLOW (T') = FOLLOW (T) = { **+,), $** }

FOLLOW (F) = FOLLOW (T') **U** FIRST (T') = {+,), $} **U** {*} = { **+,*,), $** }

5. CONSTRUCTION OF PARSING TABLE:

Non - Terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

6. The Predictive parser on input string is id+id * id.

Stack	Input	Output
$E	id + id * id $	
$E'T	id + id * id $	$E \rightarrow TE'$
$E'T'F	id + id * id $	$T \rightarrow FT'$
$E'T'id	id + id * id $	$F \rightarrow id$
$E'T'	+ id * id $	
$E'	+ id * id $	$T' \rightarrow \varepsilon$
$E'T+	+ id * id $	$E' \rightarrow +TE'$
$E'T	id * id $	
$E'T'F	id * id $	$T \rightarrow FT'$
$E'T'id	id * id $	$F \rightarrow id$
$E'T'	* id $	
$E'T'F*	* id $	$T' \rightarrow \varepsilon$
$E'T'F	id $	
$E'T'id	id $	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

7. Consider the following LL (1)

grammar S → i E t S | i E t S e S | a

E → b **Find the parsing table for the above grammar.**

Solution:

The given LL (1) grammar is

S → i E t S | i E t S e S | a

E → b

1. ELIMINATION OF LEFT FACTORING:

S → i E t S S' | a

S' → e S | ε

E → b

2. COMPUTATION OF FIRST: FIRST(S)

= { **i , a** }

FIRST(S') = { **e ,** ε }

FIRST (E) = { **b** }

3. COMPUTATION OF FOLLOW:

FOLLOW(S) = {**$**} **U** FIRST(S') = {$} **U** {e} = { **$, e** }

FOLLOW (S') = FOLLOW(S) = { **$, e** }

FOLLOW (E) = { **t** }

4. CONSTRUCTION OF PARSING TABLE:

Non-Terminal	INPUT SYMBOL					
	a	b	e	i	t	S
S	S → a			S → iEtSS'		
S'			S' → e S S' → ε			S' → ε
E		E → b				

8. Explain the LR parsing algorithm in detail. (11 marks)(NOV 2011, 2012)(MAY 2012)

Bottom-up syntax analysis technique can be used to parse a large class of context-free grammars. The technique is called ***LR (k) parsing.***

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

LR parsing is attractive for a variety of reasons.

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general non backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
4. An LR parser can detect a syntactic error as soon as it is possible to do a left-to-right scan of the input.

The principal drawback of the method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A special tool – an ***LR parser generator***.

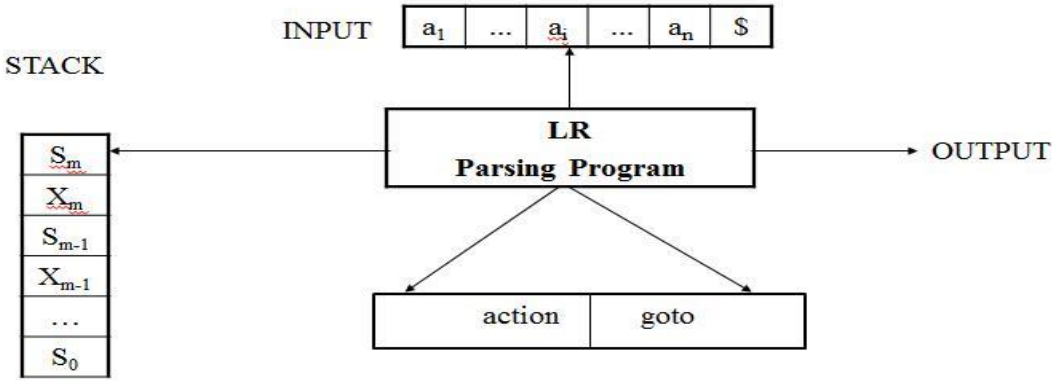
Three techniques are used for constructing an LR parsing table for a grammar.

1. The first method, called ***simple LR (SLR),*** is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed.
2. The second method, called ***canonical LR (CLR),*** is the most powerful, and the most expensive.
3. The third method, called ***look ahead LR (LALR)***, is intermediate in power and cost between the other two. The LALR method will work on most programming language grammars and, with some effort, can be implemented efficiently.

The LR Parsing Algorithm:

LR parsing consists of

- an input,
- an output,
- a stack,
- a driver program, and
- a parsing table that has two parts (***action and goto***).



Model of an LR Parser

- The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.
- The **parsing program** reads characters from an input buffer one at a time.
- The program uses a **stack** to store a string of the form $s_0X_1s_1X_2s_2 \ldots X_ms_m$, where, s_m is on top.
- Each X_i is a grammar symbol and each s_i is a symbol called a ***state.***
- Each state symbol summarizes the information contained in, the stack below it, and the combination of the state symbol on top of the' stack and' the current input symbol are used to index the parsing table and determine the shift reduce parsing decision.

The parsing table consists of two parts,

- a parsing action function ***action*** and
- a goto function ***goto***.

- The program driving the LR parser behaves as follows.
- It determines s_m, the state currently on top of the stack, and a_i, the current, input symbol.
- It then consults $action[s_m, a_i]$, the parsing action table entry for state s_m and input a_i, which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$
3. accept, and
4. error

LR PARSING ALGORITHM:

Input: An input string ***w*** and an LR parsing table with functions ***action*** and ***goto*** for a grammar G.

Output: If w is in L (G), a bottom-up parse for w; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and ***w$*** in the input buffer. The parser then executes the program until an accept or error action is encountered.

set *ip* to point to the first symbol of
w$; **repeat forever begin**

let s bet the state on the top of the stack
and a the symbol pointed to by ip;

if action[s,a]=shift s' **then begin** push a
then s' on top of the stack; advance
ip to the next input symbol

end

else if action[s,a]=reduce A $\rightarrow$ β **then**
begin pop 2*| β| symbols off the stack;
let s' be the state now on top of the stack;
push A the goto[s',A] on top of the stack;
output the production A $\rightarrow$ β

end

else if action[s,a]=accept **then**
return

else *error*()

end

9. Consider the following grammar to construct the SLR parsing table (11 marks) (NOV 2012)

E → E+T | T

T → T*F | F

F → (E) | id

Construct an LR parsing table for the above grammar. Give the moves of the LR parser on id * id + id.

Solution:

The given SLR grammar is

E -> E + T / T

T -> T * F / F

F -> (E) / id

Let the grammar G be

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> id

1. **The augmented Grammar G':**

E' -> E

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> id

2. **COMPUTATION OF CLOSURE FUNCTION: I_0 :**

E' -> .E

E -> .E + T

E -> .T

T -> . T * F

T -> .F

F -> .(E)

F -> . id

3. COMPUTATION OF GOTO FUNCTION:

goto (I, X) { **where I -> set of states and X -> E, T, F, +, *, (,), id** }

goto(I_0, E)

I_1: E' -> E.
E -> E. + T

goto(I_0, T)

I_2: E -> T.
T -> T. * F

goto(I_0, F)

I_3: T -> F.

goto(I_0, +) -> NULL

goto(I_0, +) -> NULL

goto(I_0, ( )

I_4 : F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> .id

goto(I_0,)) -> NULL

goto(I_0, id)

I_5: F -> id.

Repeat in the new set for the closure function

goto(I_1, +)

I_6: E -> E + .T T ->
.T * F T ->
.F
F -> .(E)
F -> .id

goto(I_2, *)

I_7: T -> T * .F F ->

.(E) F ->

.id

goto(I_3, X) -> NULL

goto(I_4, E)

I_8: F -> (E .)

E -> E . + T

goto(I_4, T)

I_2: E -> T.

T - > T.* E

goto(I_4, F)

I_3: T -> F.

goto(I_4, ( )

I_4: F -> (.E)

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

goto(I_4, id)

I_5: F -> id.

goto(I_6, T)

I_9: E -> E + T.

T -> T. * F

goto(I_6, F)

I_3: T -> F.

goto(I_6, ( )

I_4: F -> (.E)

E -> **.**E + T

E -> **.**T

T -> **.** T * F

T -> **.**F

F -> **.**(E)

F -> **.** id

goto(I_6, id)

I_5: F -> id**.**

goto(I_7, F)

I_{10}: T -> T * F**.**

goto(I_7, ( )

I_4: F -> (.E)

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

goto(I_7, id)

I_5: F -> id.

goto(I_8,))

I_{11}: F -> (E).

goto(I_8, +)

I_6: E -> E + **.**T T ->

.T * F T ->

.F

F -> **.**(E)

F -> **.**id

goto(I_9, *)

I_7: T -> T *.F F ->

.(E)

F -> .id

4. CONSTRUCTION OF PARSING TABLE:

1. Shifting Process

I_0 : goto(I_0, E) = I_1

goto(I_0, T) = I_2

goto(I_0, F) = I_3

goto(I_0, ( ) = I_4

goto(I_0, id) = I_5

I_1: goto(I_1, +) = I_6

I_2: goto(I_2, *) = I_7

I_4: goto(I_4, E) = I_8

goto(I_4, T) = I_2

goto(I_4, F) = I_3

goto(I_4, ( ) = I_4

goto(I_4, id) = I_5

I_6: goto(I_6, T) = I_9

goto(I_6, F) = I_3

goto(I_6, ( ) = I_4

goto(I_6, id) = I_5

I_7: goto(I_7, F) = I_{10}

goto(I_7, ( ) = I_4

goto(I_7, id) = I_5

I_8: goto(I_8,)) = I_{11}

goto(I_8, +) = I_6

I_9: goto(I_9, *) = I_7

2. i) ELIMINATION OF LEFT RECURSION ELIMINATION

E -> TE'

E' -> +TE' / ε

T -> FT'

T' -> * FT' / ε

F -> (E) / id

ii) COMPUTATION OF FIRST

FIRST (E) = FIRST (T) = FIRST (F) = { **(, id** }

FIRST (E`) = { **+, ε** }

FIRST (T) = FIRST (F) = { **(, id** }

FIRST (T`) = { ***, ε** }

FIRST (F) = { **(, id** }

iii) COMPUTATION OF FOLLOW

FOLLOW (E) = {**$**} **U** FOLLOW (E) = { **), $** }

FOLLOW (E') = FOLLOW (E) = { **), $** }

FOLLOW (T) = FOLLOW (E') **U** FIRST (E') = {), $} **U** {+} = { **+,), $** }

FOLLOW (T') = FOLLOW (T) = { **+,), $** }

FOLLOW (F) = FOLLOW (T') **U** FIRST (T') = {+,), $} **U** {*} = { **+,*,), $** }

3. Reducing Process

I_2:	E -> T.	**FOLLOW (T)**	= { **+ ,) , $** }
I_3:	T -> F.	**FOLLOW (F)**	= { **+ , * ,) , $** }
I_5:	F -> id.	**FOLLOW (F)**	= { **+ , * ,) , $** }
I_9:	E -> E + T.	**FOLLOW (T)**	= { **+ ,) , $** }
I_{10}:	T -> T * F.	**FOLLOW (F)**	= { **+ , * ,) , $** }
I_{11}:	F -> (E).	**FOLLOW (F)**	= {**+ , * ,) , $** }

5. SLR Parsing Tables of Expression Grammar:

State	Action						Goto		
	id	+	*	(	)	S	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

6. The SLR parser and given input string is id+id.

S.No	STACK	I/P STRING	PARSING ROUTINE
(1)	**0**	**id + id $**	action[0, id] = s5 then **shift 's5'** push **id** and **5** into the stack
(2)	**0 id 5**	**+ id $**	action [5, +] = r6 then **reduce r6:** **F -> id** 1) POP **2 symbols** from the stack 2) goto [0, F] = **3** 3) Push **'F3'** into the stack
(3)	**0F3**	**+id$**	action[3,+] = r4 then **reduce r4:** **T->F** 1) POP **2 symbols** from the stack 2) goto [0, T] = **2** **3)** Push **'T2'** into the stack
(4)	**0T2**	**+id$**	action[2,+] = r2 then **reduce r4:** **E->T** 1) POP **2 symbols** from the stack 2) goto [0, E] = **1** 3) Push **'E1'** into the stack

(5)	**0E1**	**+id$**	action[1, +] = s6 then **shift 's6'** push **+** and **6** into the stack
(6)	**0E1+6**	**id$**	action[6, id] = s5 then **shift 's5'** push **id** and **5** into the stack
(7)	**0E1+6id5**	**$**	action[5,$] = r6 then **reduce r6:** **F->id** 1) POP **2 symbols** from the stack 2) goto [6, F] = **3** 3) Push **'F3'** into the stack
(8)	**0E1+6idF3**	**$**	action[3,$] = r4 then **reduce r4:** **T->F** 1) POP **2 symbols** from the stack 2) goto [6, T] = **4** 3) Push **'T4'** into the stack
(9)	**0E1+6T4**	**$**	action[4,$] = r1 then **reduce r1:** **E->E+T** 1) POP **6 symbols** from the stack 2) goto [0, E] = **1** 3) Push **'E1'** into the stack
(10)	**0E1**	**$**	action[1,$] = **acc** **The given input string is accepted.**

10. List out and discuss the different type of intermediate code? (11 marks) (NOV 2012)

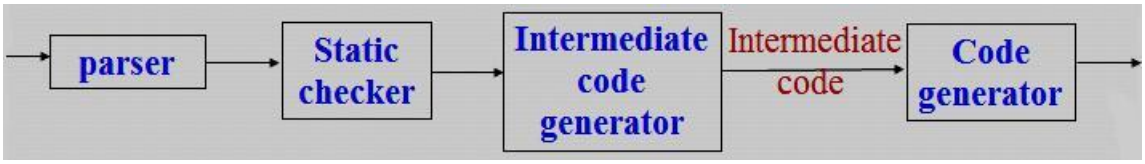
In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.

A source program can be translated directly into target language, some benefits of using machine-independent intermediate form:

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

It is used to translates into an intermediate form programming language constructs such as

- Declaration
- Assignment statements
- Boolean Expression
- Flow of control statements



The three kinds of intermediate representations are

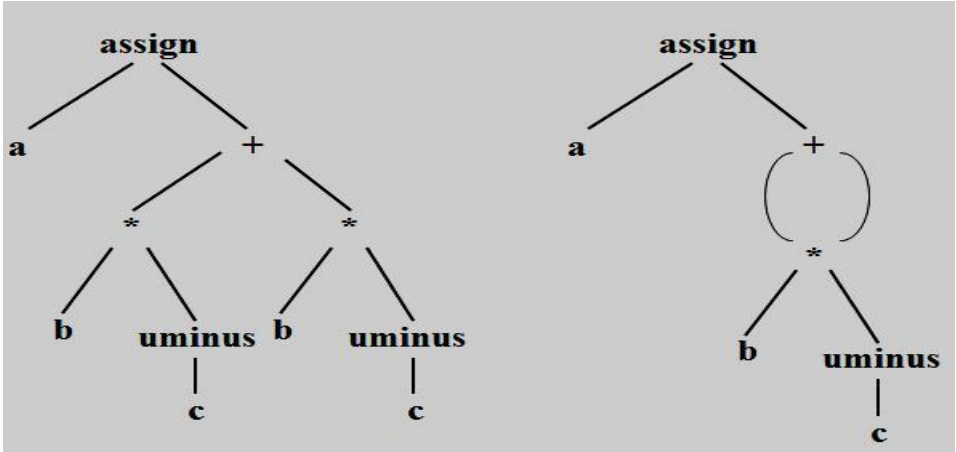
i. ***Syntax trees***
ii. ***Postfix notation***
iii. ***Three - address code***

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

Graphical representations:

- A ***syntax tree*** depicts the natural hierarchical structure of a source program.
- A ***DAG (Directed Acyclic Graph)*** gives the same information but in a more compact way because common sub-expressions are identified.

A syntax tree and DAG for the assignment statement ***a := b * - c + b * - c***



(a) Syntax tree **(b) DAG**

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

Syntax tree directed-translation:

- Syntax trees for assignment statements are produced by the syntax-directed definition.
- Non-terminal S generates an assignment statement.

The syntax-directed definition will produce the dag if the functions

- ***mkunode(op, child)***
- ***mknode(op, left, right)***
- ***mkleaf(id,id.place)***

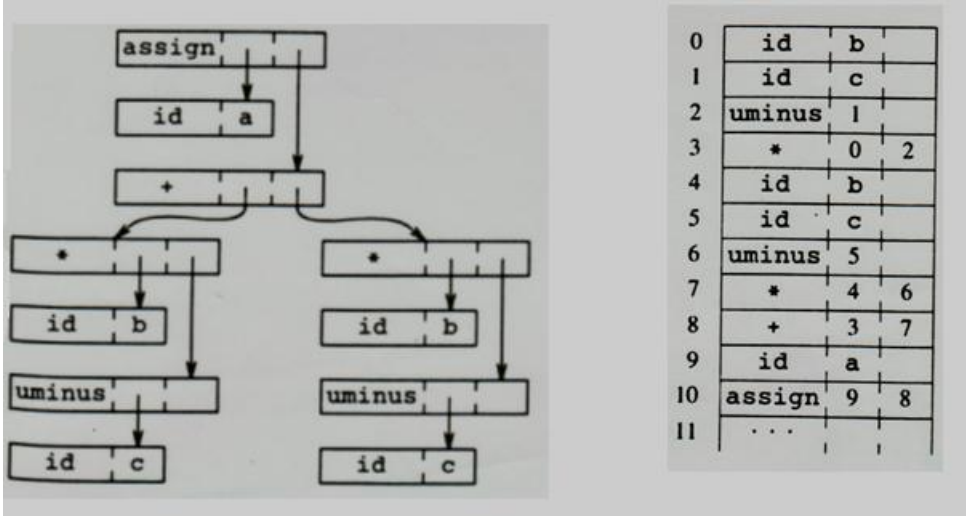
- return a pointer to an existing node whenever possible, instead of constructing new nodes.
- The token ***id*** has an attribute place that points to the symbol-table entry for the identifier.
- The symbol table entry can be found from an attribute ***id.name***, representing the lexeme associated with that occurrence of ***id***.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	S.nptr := mknode ('assign', mkleaf (id, id.entry), E.nptr)
$E \rightarrow E_1 + E2$	E.nptr := mknode('+', E_1.nptr,E_2.nptr)
$E \rightarrow E_1 * E2$	E.nptr := mknode('*', E_1.nptr,E_2.nptr)
$E \rightarrow - E_1$	E.nptr := mknode('uminus',E_1.nptr)
$E \rightarrow (E1)$	E.nptr := E_1.nptr
$E \rightarrow id$	E.nptr := mkleaf(id, id.entry)

Syntax-directed definition to produce syntax tress for assignment statements

Two way representation of syntax trees:

- Each node is represented as a record with a field for its operator and additional fields for pointers to its children.
- Nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.
- All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.



Two way representation of syntax trees

11. Explain the types of Three-address statements? (6 marks)

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 fixed or floating-point arithmetic operator, or a logical operator on boolean-valued data.

Thus a source language expression like **x + y * z** might be translated into a sequence

$$t_1 := y * z$$

$$t_2 := x + t_1$$

where t_1 and t_2 are compiler-generated temporary names.

- Three-address code is 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 are represented by the three-address code sequences

The three address codes for the following ***a: = b * -c + b * -c***

$t_1 := - c$	$t_1 := - c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := - c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 +$	
t_4 a := t_5	
(a) Code for syntax tree	***(b) Code for DAG***

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

Types of Three-Address Statements:

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ”back patching”.

The common three-address statements are

1. ***Assignment statements*** of the form ***x: = y op z***, where op is a binary arithmetic or logical operation.
2. ***Assignment instructions*** of the form ***x: = op y,*** where op is a unary operation.
   - The unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. ***Copy statements*** of the form ***x: = y*** where the value of y is assigned to x.
4. ***The unconditional jump*** goto L. The three-address statement with label L is the next to be executed.
5. ***Conditional jumps*** such as ***if x relop y goto L.***
   - This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next.
6. **param x and call p, n** for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of three-address statements

   param x1

   param x2

   param xn

   call p, n

   generated as part of a call of the procedure $p(x_1,, x_2,..., x_n)$. The integer n indicating the number of actual parameters in ”call p, n” is not redundant because calls can be nested.
7. **Indexed assignments** of the form **x: = y[i]** and **x [i]: = y**.
   - The first of these sets x to the value in the location i memory units beyond location y. The statement x[i]:=y sets the contents of the location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects.
8. **Address and pointer assignments** of the form **x: = &y, x: = *y** and ***x: = y**.

   The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object. In the statement x: = *y, presumably y is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, *x: = y sets the r-value of the object pointed to by x to the r-value of y.

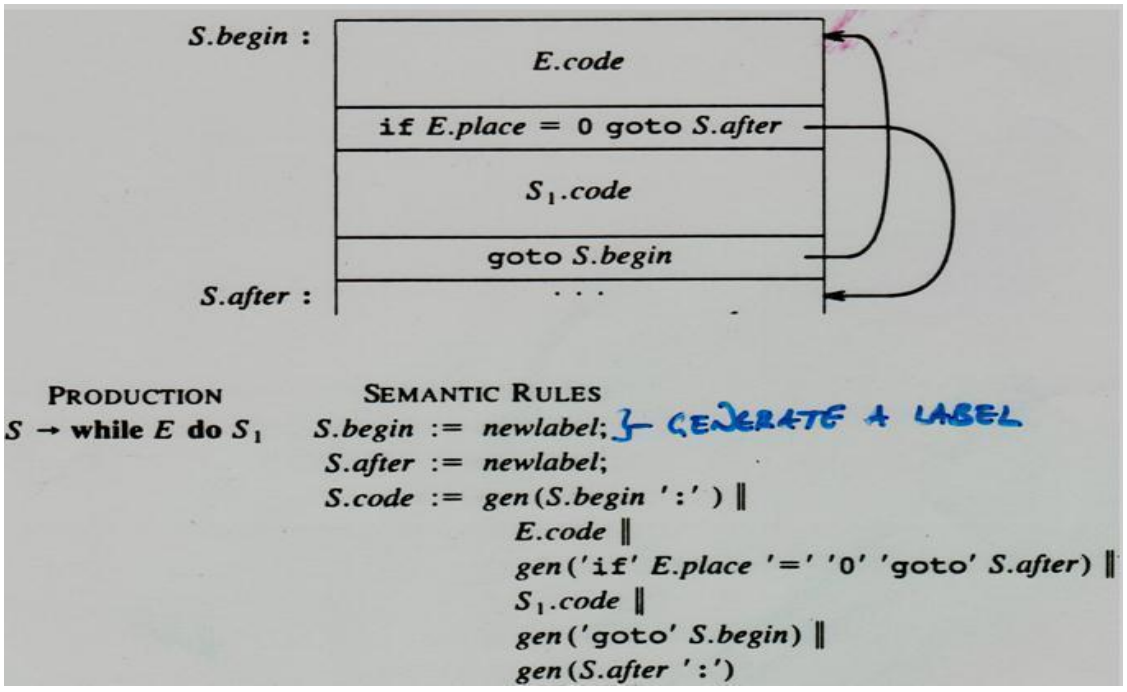
12. Explain the syntax directed translation into three- address code? (5 marks)

- When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal E on the left side of ***E → E1 + E*** will be computed into a new temporary t.
- In general, the three- address code for ***id: = E*** consists of code to evaluate E into some temporary t, followed by the assignment ***id.place: = t.***
- If an expression is a single identifier, say y, then y itself holds the value of the expression.
- We create a new name every time a temporary is needed; techniques for reusing temporaries are given.
- The ***S-attributed definition*** generates three-address code for assignment statements.
- Given input a: = b* – c + b* – c, it produces the code.
- The synthesized attribute ***S.code*** represents the three- address code for the assignment S.
- The non-terminal E has two attributes:
    1. ***E.place,*** the name that will hold the value of E, and
    2. ***E.code,*** the sequence of three-address statements evaluating E.
- The function ***newtemp*** returns a sequence of distinct names t1, t2,... in response to successive calls.
- Use the notation ***gen(x ': = ' y '+' z)*** to represent the three-address statement ***x: = y + z.***
- Expressions appearing instead of variables like x, y, and z are evaluated when passed to ***gen***, and quoted operators or operands, like '+', are taken literally. Three- address statements might be sent to an output file, rather than built up into the code attributes.
- Flow-of-control statements can be added to the language of assignments by productions and semantic rules like the ones for while statements.
- The code for ***S → while E do S1***, 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.
- ***S.after*** 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 F becomes zero, control leaves the while statement.

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

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	S.code := E.code || **gen**(**id**.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place := newtemp ; $E.code := E_1.code \|\| E_2.code \|\|$ **gen**(E.place ':=' E_1.place **'+'** E_2.place)
$E \rightarrow E_1 * E_2$	E.place := newtemp ; $E.code := E_1.code \|\| E_2.code \|\|$ **gen**(E.place ':=' E_1.place **'*'** E_2.place)
$E \rightarrow - E_1$	E.place:= newtemp ; E.code := E_1.code || **gen**(E.place ':=' **'uminus'** E_1.place)
$E \rightarrow (E_1)$	E.place:= E_1.place ; E.code := E_1.code
$E \rightarrow id$	E.place := id.place ; E.code := ''

- Expressions that govern the flow of control may in general be Boolean expressions containing relational and logical operators.
- The semantic rules for while statements to allow for flow of contro1 within Boolean expressions



Semantic rules generating code for a while statement

13. Explain the three implementation of three-address statements? (11 marks)

Implementations of three-Address Statements

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

(i) Quadruples:

- A ***quadruple*** is a record structure with four fields, are ***op, arg l, arg 2,*** and ***result.***
- The op field contains an internal code for the operator.
- The three-address statement ***x: = y op z*** is represented by placing y in arg 1, z in arg 2, and x in result. Statements with unary operators like ***x: = – y or x: = y*** do not use ***arg 2***.
- Operators like ***param*** use neither arg2 nor result.
- Conditional and unconditional jumps put the target label in result.
- The quadruples for the assignment ***a: = b * -c + b * -c***

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

	op	arg1	arg2	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

Quadruples of three address statements

- The contents of field's ***arg 1***, ***arg 2,*** and ***result*** are normally pointers to the symbol-table entries for the names represented by these fields.
- If so, temporary names must be entered into the symbol table as they are created.

(ii) Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- Three-address statements can be represented by records with only three fields: ***op, arg 1*** and ***arg2***.
- The field's ***arg l*** and ***arg2,*** for the arguments of ***op***, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as ***triples.***
- Triples correspond to the representation of a syntax tree or dag by an array of nodes.

The assignment a := b * -c + b * -c

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

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Triple representation of three – address statements

- Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves.
- The information needed to interpret the different kinds of entries in the ***arg 1*** and ***arg2*** fields can be encoded into the op field or some additional fields.
- The copy statement ***a: = t_5*** is encoded in the triple representation by placing ***a*** in the ***arg 1*** field and using the operator assign.
- A ternary operation like ***x[i]: = y*** and ***x: = y[i]*** requires two entries in the triple structure

	op	*arg1*	*arg2*
(0)	[ ] :=	x	i
(1)	assign	(0)	y

x[i] := y

	op	*arg1*	*arg2*
(0)	[ ] :=	y	i
(1)	assign	x	(0)

x := y[i]

(iii) Indirect Triples:

- Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called ***indirect triples.***
- For example, let us use an array statement to list pointers to triples in the desired order.

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

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

Indirect triples representation of three-address statements

14. Write the quadruple representation for the assignment statement a : = - b * (c + d)

(5 marks)(MAY 2012)

- A ***quadruple*** is a record structure with four fields, are ***op, arg l, arg 2,*** and ***result.***
- The op field contains an internal code for the operator.
- The three-address statement ***x: = y op z*** is represented by placing y in arg 1, z in arg 2, and x in result. Statements with unary operators like ***x: = – y or x: = y*** do not use ***arg 2***.
- The quadruples representation for the assignment statement ***a : = - b * (c + d)***

The quadruple representation for the assignment statement a := - b * (c + d)

$t_1 := - b$

$t_2 := c + d$

$t_3 := t_1 * t_2$

$a := t_3$

	op	arg1	arg2	result
(0)	uminus	b		t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3		a

15. Describe in detail the declarations in a procedure and the methods to keep track of scope information. (11 marks) (NOV 2013)

- The sequence of declarations in a procedure or block is examined; we can lay out storage for names local to the procedure.
- For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name.
- The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.
- When the front end generates addresses, it may have a target machine.
- Suppose that addresses of consecutive integers differ by 4 on a byte- addressable machine.
- The address calculations generated by the front end may therefore include multiplications by 4.
- The instruction set of the target machine may also favor certain layouts of data objects, and hence their addresses.

Declarations in a Procedure:

- The syntax of languages such as C, Pascal, and FORTRAN, allows all the declarations in a single procedure to be processed as a group.
- In this case, a global variable, say ***offset,*** can keep track of the next available relative address.
- Non-terminal P generates a sequence of declarations of the form ***id: T***.
- Before the first declaration is considered, offset is set to 0.
- As each new name is seen, that name is entered in the symbol table with offset equal to the current value of ***offset***, and ***offset*** is incremented by the width of the data object denoted by that name.
- The procedure ***enter (name, type, offset)*** creates a symbol-table entry for name, gives it type and relative address offset in its data area.
- We use synthesized attributes type and width for non-terminal T to indicate the type and width, or number of memory units taken by objects of that type.
- Attribute type represents a type expression constructed from the basic type's ***integer*** and ***real*** by applying the type constructors' ***pointer*** and ***array***.
- If type expressions are represented by graphs, then attribute ***type*** might be a pointer to the node representing a type expression.
- ***Integers*** have ***width 4*** and ***real*** have ***width 8***.
- The ***width of an array*** is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be ***4.***

$P \rightarrow$ D	{ *offset* := 0 }
$D \rightarrow D$; D	
$D \rightarrow$ **id** : T	{ *enter*(**id**.*name*, T.*type*, *offset*); *offset* := *offset* + T.*width* }
$T \rightarrow$ **integer**	{ T.*type* := *integer*; T.*width* := 4 }
$T \rightarrow$ **real**	{ T.*type* := *real*; T.*width* := 8 }
$T \rightarrow$ **array** [**num**] **of** T_1	{ T.*type* := *array*(**num**.*val*, T_1.*type*); T.*width* := **num**.*val* × T_1.*width* }
$T \rightarrow \uparrow T_1$	{ T.*type* := *pointer*(T_1.*type*); T.*width* := 4 }

Computing the types and relative addresses of declared names

- In Pascal and C, a pointer the type of the object pointed. Storage allocation for such types is simpler if all pointers have the same width.
- The initialization of offset in the translation scheme is the first production appears on one line as:

  P → { offset: = 0 } D

- Non-terminals generating ε, called marker non-terminals can be used to rewrite productions so that all actions appear at the ends of right sides. Using a marker non-terminal M, can be

  P → M D

  M →ε { offset: = 0 }

Keeping Track of Scope Information:

In a language with nested procedures, names local to each procedure can be assigned relative addresses. When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended.

The semantic rules to the following language

P → D

D → D; D | id: T proc id; D; S

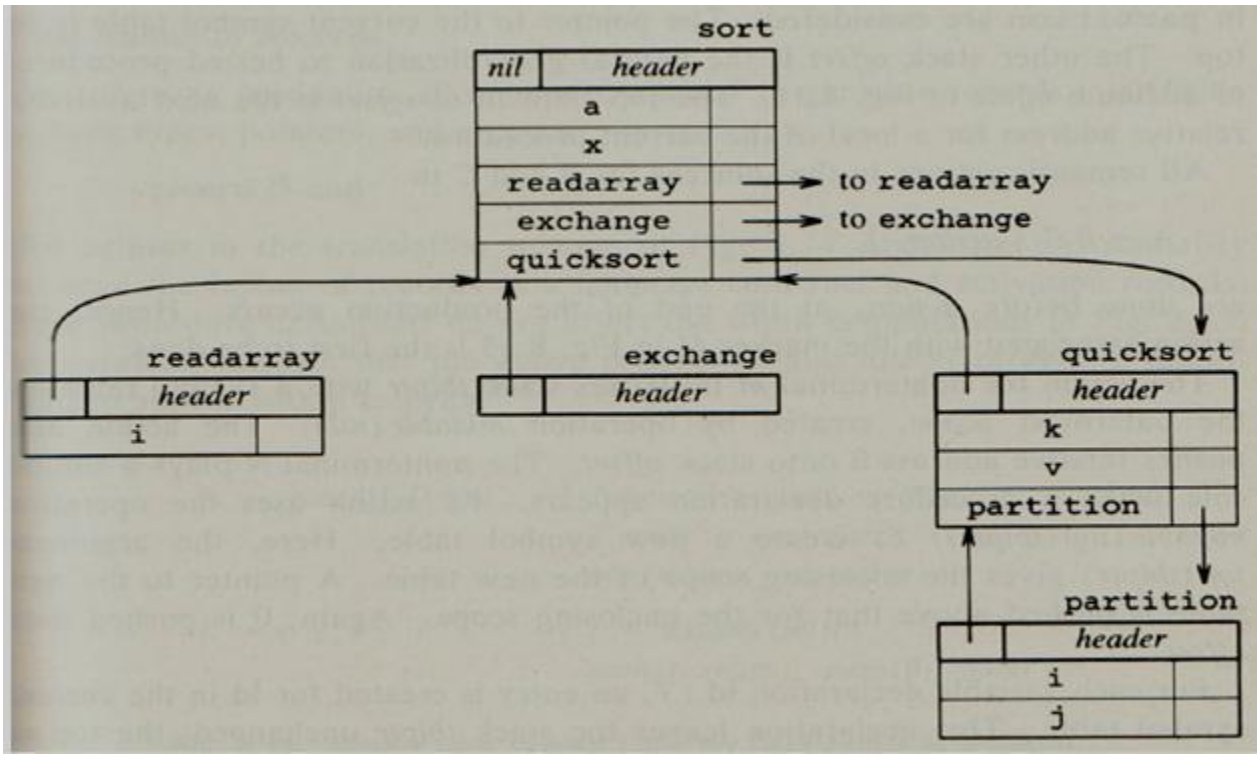
- The production for non-terminals S for statements and T for types. The non-terminal T has synthesized attributes type and width. For simplicity, suppose that there is a separate symbol table for each procedure in the language.

- A new symbol table is created when a procedure declarations **D** →**proc id D**1; **S** and entries for the declarations in D1 in a new symbol table. The new table points back to the symbol table of the enclosing procedure; the name represented by **id** itself is local to the enclosing procedure.

For example, the symbol tables for five procedures.

- The symbol tables for procedures ***readarra***y, ***exchange*** and ***quick sort*** point back and containing ***procedure sort***, consisting of the entire program.
- The ***partition*** is declared with ***quick sort, its*** table point to that of ***quick sort.***

Symbol tables for nested procedures:



The Semantic rules for operations:

1. ***mktable (previous)***
   - It creates a new symbol table and returns a pointer to the new table.
   - The argument ***previous*** points to a previously created symbol table, presumably that for the enclosing procedure.
   - The pointer ***previous*** is placed in a header for the new symbol table, along with additional information such as the nesting depth of a procedure.
   - We can also number the procedures in the order they are declared and keep this number in the header.
2. ***enter (table, name, type, offset)***
   - It creates a new entry for name ***name*** in the symbol table pointed to by table.
   - Again, ***enter*** places type ***type*** and relative address ***offset*** in fields within the entry.
3. ***addwidth (table, width)*** - records the cumulative width of all the entries table in the header associated with this symbol table.
4. ***enterproc (table, name, newtable)***
   - It creates a new entry for procedure ***name*** in the symbol table pointed to by table.
   - The argument ***newtable*** points to the symbol table for this procedure ***name.***

- The translation scheme shows how data can be in one pass, using a stack ***tblptr*** to hold pointers to symbol tables of the enclosing procedures.
- With the symbol tables to ***tblptr*** will contain pointers to the tables for ***sort, quicksort,*** and ***partition*** when the declarations in partition are considered.
- The pointer to the current symbol table is on top.
- The other stack ***offse***t is the natural generalization to nested procedures of attribute ***offset.***
- The top element of ***offset*** is the next available relative address for a local of the current procedure.

- The action for non-terminal M initializes stack ***tblptr*** with a symbol table for the outermost scope, created by operation ***mktable(nil).*** The action also pushes relative address 0 onto stack ***offset.***
- The non-terminal N plays a similar role when a procedure declaration appears.
- Its action uses the operation ***mktable(top(tblptr))*** to create a new symbol table.
- The argument ***top(tblptr)*** gives the enclosing scope of the new table.
- A pointer to the new table is pushed above that for the enclosing scope. Again, 0 is pushed onto ***offset***.

Processing declarations in nested procedures

P →M D	{ addwidth (top(tblptr), top(offset)); pop(tblptr); pop(offset) }
M → ε	{ t := mktable(nil); push(t, tblptr); push(0, offset) }
D → D1 ;D2	
D → proc id ; N D1 ;S	{ t := top(tblptr); addwidth(t. top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.name, t) }
D → id : T	{ enter(top(tblptr), **id.**name, T.type, top(offset)); top(offset) := top(offset) + T.width }
N → ε	{ t := mktable(top(tblptr)); push(t, tblptr); push(0, offset) }

Field Names in Records:

The following production allows non-terminal T to generate records in addition to basic types, pointers, and arrays:

T →record D end

T → record L D end	{ T.type := record(top(tblptr)); T.width := top(offset); pop(tblptr); pop(offset) }
L→ ε	{ t:= mktable(nil); push(t, tblptr); push (0, offset) }

16. Explain in detail about assignment statements? (11 marks)

- Expressions can be of type integer, real, array and record fields.
- The translation of assignment into three- address code that names can be looked up in the symbol table and how elements of array and records can be accessed.

Names in the Symbol Table:

- Three address statements using names for pointers to their symbol table entries.
- The lexeme for the name represented by **id**, the attribute as ***id.name***
- The operation ***lookup (id.name)***, check if there is an entry for the occurrence of the name in the symbol table.
- If so, a pointer of the entry is returned.
- Otherwise returns ***nil*** to indicate that no entry is found.

The semantic action use procedure ***emit*** to 3 address statements to an output file, code attributes for non-terminals.

$$S \rightarrow \textbf{id: = E}$$

- The non-terminal ***S*** represents the **name** modified **lookup** operation first checks if name appears in the current symbol table, accessible through **table pointer**.
- If not, lookup uses the pointer in the header of a table to find the symbol table.
- If the name cannot be found, then **lookup** returns **nil.**

Translation scheme to produce three-address code for assignments

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. placer:= 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 }

Reusing Temporary Names:

- The ***newtemp*** generates a new temporary name each time temporary is needed.
- The temporaries used to hold intermediate values in expression calculations for the symbol table and space has to be allocated to hold their values.
- Temporaries can be reused by changing ***newtemp.***
- The temporary data are generated during the syntax directed translation of expression.
- The code generated by the rules is ***E → E1+E2*** of the form
  - evaluate E1 into t1
  - evaluate E2 into t2
  - $t := t_1 + t_2$

Consider the assignment statements **x: = a*b + c*d – e*f**

STATEMENTS	**VALUE OF C**
	0
$0 := a * b ;	1 c incremented by 1
$1 := c * d ;	2 c incremented by 1
$0 := $0 + $1 ;	1 c decremented twice, incremented once
$1 := e * f ;	2 c incremented by 1
$0 := $0 - $1 ;	1 c decremented twice, incremented once
x := $0 ;	0 c decremented once

- A count c, initialized to zero.
- Whenever a temporary name is used as an operand, decrement c by 1.
- Whenever a new temporary name is created, use $c and increment c by 1.

Type Conversion with Assignments:

- There are different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate the type conversion instructions.
- Consider the grammar for assignment statements, there are two types –***real and integer***, with integers converted to reals when necessary.
- An attribute **E.type** holds the type of an expression which is either ***real or integer.***

The semantic rule for the ***E.type*** production E → E+E is:

E → E+E { E.type :=
if E1.type = integer ***and***
E2.type = integer ***then*** integer
else real }

For example, for the input

x := y + i * j

Assuming ***x and y*** have type ***real*** and ***i and j*** have type ***integer,*** the output

as **t_1 := i int* j**

t_2 := inttoreal t_1

t_3:= y real+ t_2 x

:= t_3

Semantic action for E → E+E E.place
:= newtemp;

if E1.type = integer and E2.type = integer **then begin**
emit(E.place ':=' E1.place 'int+' E2.place);
E.type := integer
end
e**lse** if E1.type = real and E2.type = real **then begin**
emit(E.place ':=' E1.place 'real+' E2.place);
E.type := real
end
else if E1.type = integer and E2.type = real **then**
begin u := newtemp;
emit(u ':=' 'inttoreal' E1.place);
emit(E.place ':=' u 'real+' E2.place);
E.type := real
end

else if E1.type = real and E2.type = integer **then**

begin u := newtemp;

emit(u ':=' 'inttoreal' E2.place);

emit(E.place ':=E1.place 'real+' u);

E.type := real

end

else

E.type := type_error;

Accessing Fields in Records:

- The compiler must keep track of both the types and relative addresses of the fields of a record.
- The information in the symbol table entries for the field names that looking up names tin the symbol table can be used as field names.

  pointer (record (t)) or ***p ↑ .info + 1***
- The ***type of p*** is the record (t), from which t can be extracted.
- The ***name info*** field lookup in the symbol table pointed to by t.

17. State and write the semantic rules for Boolean expressions. (11 marks)(MAY 2013)

In programming languages, Boolean expressions have two primary purposes

1. They are used to compute logical values.
2. 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.
- Relational expression of the form **E1 relop E2,** where E1 and E2 arithmetic expressions.
- Consider boolean expressions with the following grammar:

  E → E ***or*** E | E ***and*** E | ***not*** E | (E) | id ***relop*** id | ***true*** | ***false***

- We use the attribute ***op*** to determine which of the comparison operators **<, <=, =, !=, >, or >=** is represented by ***relop.***
- Assume that '***or'*** and ***'and'*** are ***left-associative***, and that ***or*** has ***lowest precedence***, then ***and,*** then ***not.***

Methods of Translating Boolean Expression:

There are two principal methods of representing the value of a Boolean expression.

1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.
2. The second principal method of implementing boolean expression is by flow of control that is representing the value of a Boolean expression by a position reached in a program. This method is implementing the Boolean expressions in flow of control statements, such as the if-then and while-do statements.

(i) Numerical Representation:

- Consider the implementation of boolean expression using ***1 to denote true*** and ***0 to denote false***.
- For example, expressions as **a or b and not c**

The translation for 3 address sequence is

t_1 := not c t_2
:= b and t_1 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.

```
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 expression

$E \rightarrow E_1$ **or** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'or' E_2.*place*) }
$E \rightarrow E_1$ **and** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'and' E_2.*place*) }
$E \rightarrow$ **not** E_1	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' 'not' E_1.*place*) }
$E \rightarrow$ (E_1)	{ *E.place* := E_1.*place* }
$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.place* := *newtemp*; *emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* 'goto' *nextstat* + 3); *emit*(*E.place* ':=' '0'); *emit*('goto' *nextstat* + 2); *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **true**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **false**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '0') }

- We assume that ***emit*** places three -address statements into output file in the right format.
- The ***nextstat*** that gives the index of the next three-address statement in the output sequence and ***emit*** increments ***nextstat*** after producing each three-address statement.
- We use the attribute **op** to determine which of the comparison operators is represented by ***relop.***

(ii) Shot-Circuit or jumping code:

- Translate a Boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called ***"short-circuit" or "jumping" code.***
- It is possible to evaluate boolean expressions without generating code for the boolean operators ***and, or*** and ***not***, the value of an expression by a position in the code sequence.

Translation of a<b or c<d and e<f

The three address code as

100: ***if*** a< b goto ***103***

101: t_1 := 0

102: goto ***104***

103: t_1 := 1

104: ***if*** c< d goto ***107***

105: t_2 := 0

106: goto ***108***

107: t_2 := 1

108: ***if*** e< f goto ***111***

109: t_3 := 0

110: goto ***112***

111: t_3 := 1

112: t_4 := t_2 ***and*** t_3

113: t_5 := t_1 ***or*** t_4

(iii) Flow-of-Control Statements:

Consider the translation of boolean expressions into three-address code as if-then, if-then-else, and while –do statements such those generated by the following grammar

S → if E then S_1

| if E then S_1 else S_2

| while E do S_1

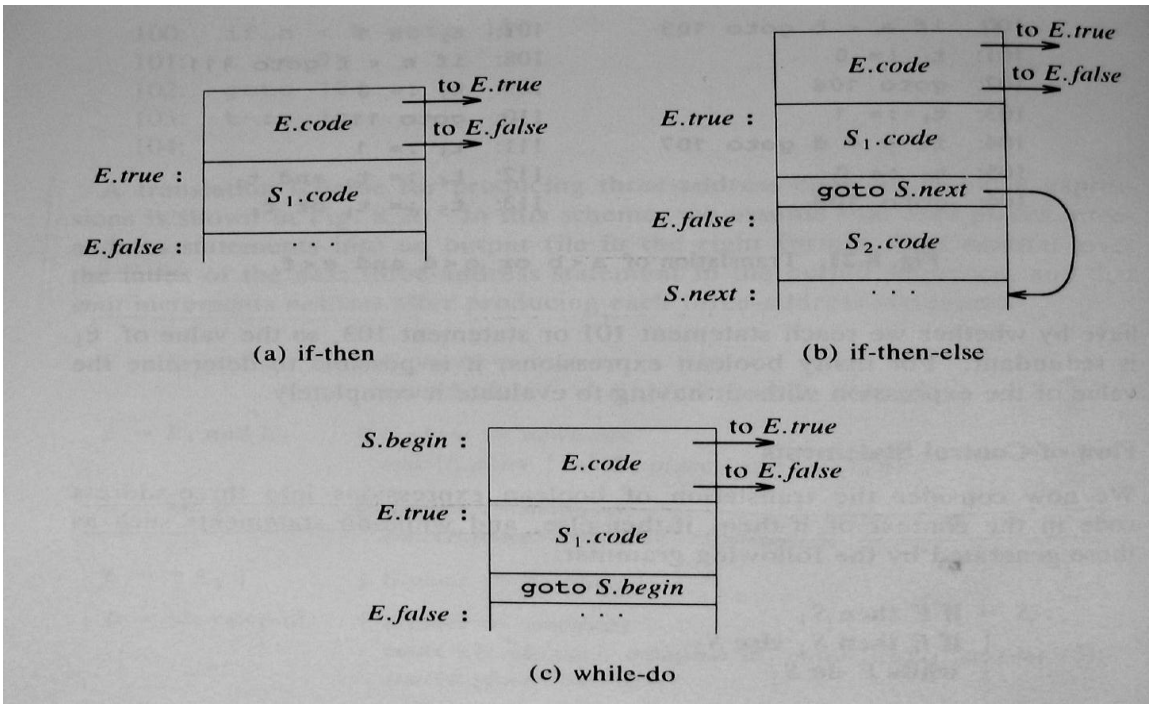
In the translation, we assume that a three-address code statement can have a symbolic label, and that the function newlabel generates such labels.

With a boolean expression E, we associate two labels:

- ***E.true,*** the label to which control flows if E is true.
- ***E.false,*** the label to which control flows if E is false.

We associate to S the inherited attribute ***S.next*** that represents the label attached to the first statement after the code for S.

Code for if-then, if-then-else, and while-do statements:



Syntax-directed definition for flow-of control statements:

PRODUCTION	SEMANTIC RULES
S → 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 → 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 → 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)

(iv) Control Flow Translation of Boolean Expression:

Boolean Expressions are translated in a sequence of conditional and unconditional jumps to either ***E.true*** or ***E.false.***

- ***E.true***, the place control is to reach if ***E is true*** and
- ***E.false,*** the place control is to reach if ***E is false.***

The expression E is of the form a < b. The generated code is of the form

if a < b then goto E.true

goto E.false

Suppose E is of the form **E → E_1 or E_2.**

- If E_1 is true, then E is true, so E_1.true = E.true.
- If E_1 is false, then E2 must be evaluated, so E1.false is set to the label of the first statement in the code for E2.

Syntax-directed definition to produce three-address code for booleans:

PRODUCTION	SEMANTIC RULES
E → E_1 or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ' :') || E_2.code
E → E_1 and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ' :') || E_2.code
E → not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code
E → (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code
E → id_1 relop id_2	E.code := gen('if' **id_1.place** relop.op **id_2.place** 'goto' E.true) || gen('goto' E.false)
E → true	E.code := gen('goto' E.true)
E → false	E.code := gen('goto' E.false)

(v) Mixed mode Boolean expression:

- Boolean Expressions often contain arithmetic sub-expressions as in (a + b) <c.
- In languages where false has the numerical value 0 and true the value 1, then (a<b) + (b<a) can be an arithmetic expression with value 0 if a and b have the same value and 1 otherwise.

Consider the following Grammar:

E → E+E | E and E | E relop E | id

- ***E + E,*** produces an arithmetic result, and the arguments can be mixed; while expressions ***E and*** E, and ***E relop E,*** produces boolean values represented by flow of control.
- Expression ***E and E*** requires both arguments to be boolean, but the operations ***+*** and ***relop*** take either type of argument, including mixed ones.
- **E→id** is assumed of type arithmetic.
- To generate code we use a synthesized attribute ***E.type*** that will be either ***arith or bool.***
- E will have inherited attributes ***E.true*** and ***E.false*** for boolean expressions and synthesized attribute E.place for arithmetic Expressions. useful for the jumping code.

The semantic rule for **E → E_1+E_2**

E.type := arith;

if E_1.type := arith and E_2.type := arith **then begin**

E.place := newtemp;

E.code := E_1.code || E_2.code ||

gen(E.place' :=' E_1.place ' +' E_2.place)

end

esle if E_1.type := arith and E_2.type := bool **then begin**

E.place:= newtemp;

E_2.true := newlabel;

E_2.false := newlabel;

E.code := E_1.code || E_2.code ||

gen(E_2.true' :' E.place ' :=' E_1.place + 1)

|| gen('goto' nextstat + 1) ||

gen(E_2.false' :' E.place ' :=' E_1.place)

18. Explain the case statements? (5 marks)

The ***"switch" or "case"*** statement is available in various languages; the FORTRAN computed and assigns goto's for switch statements.

The switch-statement syntax

is **switch** expression

begin

case value: statement

case value: statement

...

case value: statement

default: statement

end

There is a selector expression, which is to be evaluated, followed by n constant values that the expression, including a ***default "value",*** which always the expression if no other values does.

The translation of a switch code is

1. Evaluate the expression.
2. Find which value in the list of cases is same as the value of expression. The default value matches the expression if none of the values explicitly.
3. Execute the statement associated with the value found.

- To implement a sequence of conditional **goto's** is to create a table of pair, each pair consisting of a value and a label for the code of the corresponding statement.
- A compiler to compare the value of expression with each value in the table.
- If no other match is found, the last (default) entry is sure to match.

Syntax directed translation of case statements:

Consider the following switch statement

switch E

begin

case V_1 : S_1

case V_2 : S_2

...

case V_{n-1}: S_{n-1}

default : S_n

end

To translate in the form the keyword **switch** generate 2 labels

- ***Test*** and ***next***
- A new temporary variable ***t.***
- Then parse the expression E, generate code to evaluate E into t.
- After processing E, generate the jump ***goto test.***

To translate in the form the keyword **case**

- Create a new label $\mathbf{L_i}$ and enter into the symbol table.
- We place on a stack, used only to store cases, a pointer to symbol table entry and value $\mathbf{V_i}$ of the case constant.
- Each statement ***case*** V_i ***:*** S_i***,*** creates label $\mathbf{L_i}$**,** followed by code for $\mathbf{S_i}$**,** followed by jump ***goto next.***
- When the keyword ***end*** terminate the body of switch statement.

Translation of case statement:

code to evaluate ***E into t***

goto test

$\mathbf{L_1}$**:** code for S_1

goto next

$\mathbf{L_2}$**:** code for S_2

goto next

.....

$\mathbf{L_{n-1}}$**:** code for S_{n-1}

goto next

$\mathbf{L_n}$**:** code for S_n

goto next

test: if t = V_1 goto L_1

if t = V_2 goto L_2

.....

if t = V_{n-1} goto L_{n-1}

goto $\mathbf{L_n}$

next:

Another Translation of case statement

	Code to evaluate ***E into t***
	if t ≠ V_1 goto L_1
	code for S_1
	goto next
L_1:	if t ≠ V_2 goto L_2
	code for S_2
	goto next
L_2:	...
L_{n-2}:	if t ≠ V_{n-1} goto L_{n-1}
	code for S_{n-1}
	goto next
L_n:	code for S_n
next:	

19. How the code is generated for procedure calls? (5 marks) (NOV 2011)

- The procedure is such an important and frequently used programming construct that 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.

  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 arguments of the called procedure must be evaluated and available to the called procedure in 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, location to which the called routine must transfer after it is finished.
- The return address is usually the location of the instruction that follows that call for the calling procedure.
- Finally, a jump to the beginning of the code for the called procedure must be generated.

Return Statements:

- 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.
- No exact division of run-time tasks between the calling and called procedure.

Translation includes

- Calling sequence → actions taken on entry to and exit from each procedure.
- Arguments are evaluated and put in a known places(return address) location to which the called routine must transfer after it is finished.
- Static allocation → return address is placed after code sequence itself.
- Parameters passed by reference.
- 3 address code → generates statements needed to evaluate those arguments that are simple names then the list.

For separate evaluation:

- Save E.place for each expression E in id (E, E, E, E,..)
- Data structure used is queue.

Semantics:

1) S → call id (Elist)

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

2) Elist → Elist, E

{ append E.place to end of queue }

3) Elist → E

{ initialize queue to contain only E.place }

Queue is emptied & single pointer is given to symbol table denoting value of E.

21. How back patching can be used to generate code for Boolean expressions? (11 marks) (NOV 2011)

- To implementing syntax-directed definitions, to use two passes.
- The main problem with generating codes 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 jump statements are generated.
- By generating a series of branching statement with the targets of the jumps 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. This subsequent filling of addresses for the determined labels is called

  Back patching.
- Back patching can be used to generate code for Boolean expression and flow of control statements in one pass.

To generate quadruples into a quadruple array and labels are indices to this array. To manipulate list if 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.

(i) Boolean Expressions:

Construct a translation scheme for producing quadruples for Boolean expressions during bottom-up parsing. We insert a marker non-terminal M into the grammar, the index of the next quadruple to be generated.

The grammar is:

E → E1 **or** M E2

E → E1 a**nd** M E2

E → **not** E1

E → (E1)

E → id1 **relop** id2

E → **false**

E → t**rue**

M → **ε**

- Synthesized attributes ***truelist*** and ***falselist*** of non-terminal E are used to generate jumping code for Boolean expressions.
- ***E.truelist :*** Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for ***E:=true.***
- ***E.falselist*** : Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for ***E:=false.***
- The variable ***nextquad*** holds the index of the next quadruple to follow.
- ***M.quad*** represents records the number of first statement (index). Consider the production **$E \rightarrow E_1$ and M E_2**.

The semantic actions as

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ or M E_2	{ backpatch(E_1.falselist, M.quad) E.truelist = merge(E_1.truelist, E_2.truelist) E.falselist = E_2.falselist }
$E \rightarrow E_1$ and M E_2	{ backpatch(E_1.truelist, M.quad) E.truelist = E_2.truelist E.falselist = merge(E_1.falselist, E_2.falselist) }
$E \rightarrow$ not E_1	E.truelist = E_1.falselist E.falselist = E_1.truelist
$E \rightarrow (E_1)$	E.truelist = E_1.truelist E.falselist = E_1.falselist
$E \rightarrow id_1$ 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 ___)
E → true	E.truelist = makelist(nextquad) emit(goto ___)
E → false	E.falselist = makelist(nextquad) emit(goto ___)
M → ε	M.Quad = nextquad

(ii) Flow-of –Control Statements:

- Backpatching can be used to translate flow-of-control statements in one pass. Translation scheme for statements generated by the following grammar:

S→ if E then S

| if E then S else S

| while E do S

| begin L

end | A

L → L;S

| S

Here ***S*** denotes a statement, ***L*** a statement list, ***A*** an assignment statement, and ***E*** a boolean expression.

Scheme to implement the Translation:

(1) S → if E then M S_1

{ backpatch (E.truelist , M.quad);

S_1.nextlist := mergelist (E.falselist , S_1.nextlist) }

(2) 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 := mergelist (S1..nextlist, mergelist (N.nextlist , S_2.nextlist)) }

We backpatch the jumps when E is true to the quadruple M_1.quad, which is the beginning of the code for S_1. Similarly, we backpatch 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.

(3) S → while M_1 E do M_2 S_1

{ backpatch (S_1.nextlist , M_1.quad);

backpatch (E.truelist , M_2.quad);

S.nextlist := E.falselist

emit('goto' M_1.quad) }

(4) S → begin L end

{ S.nextlist := L.nextlist }

(5) S → A

{ S.nextlist := nil }

(6) L → L_1 M S

{ backpatch(L_1.nextlist , M.quad);

L.nextlist := S.nextlist }

(7) N → ε

{ N.nextlist := makelist (nextquad);

emit('goto_') }

(8) M → ε

{ M.quad := nextquad }

UNIVERSITY QUESTIONS

2 MARKS

1. What is the use of context free grammar? **(NOV 2011) (Ref.Qn.No.7, Pg.no.3)**
2. Draw the dag for the assignment statement: a: = b * -c + b * -c **(NOV 2011) (Ref.Qn.No.38, Pg.no.8)**
3. Define Ambiguous.**(MAY 2012) (Ref.Qn.No.12, Pg.no.4)**
4. What is Parsing Tree?**(MAY 2012) (Ref.Qn.No.10, Pg.no.3)**
5. Define Three-Address Code.**(NOV 2012) (Ref.Qn.No.41, Pg.no.9)**
6. Differentiate phase and pass.**(NOV 2012) (Ref.Qn.No.32, Pg.no.7)**
7. Derive the first and follow for the follow for the following grammar.

   $S \rightarrow 0|1|AS0|BS0 \qquad A \rightarrow \varepsilon \quad B \rightarrow \varepsilon$ **(MAY 2013) (Ref.Qn.No.58, Pg.no.13)**
8. State the function of an intermediate code generator.**(MAY 2013) (Ref.Qn.No.33, Pg.no.7)**
9. Briefly describe the LL (k) items.**(NOV 2013) (Ref.Qn.No.19, Pg.no.5)**
10. What are the different forms of Intermediate representations?**(NOV 2013) (Ref.Qn.No.35, Pg.no.8)**

11 MARKS

NOV 2011(REGULAR)

1. Explain the LR parsing algorithm in detail. **(Ref.Qn.No.8, Pg.no.32)**

   (OR)
2. a) How back patching can be used to generate code for Boolean expressions? (6)

   (Ref.Qn.No.21, Pg.no.72

   b) How the code is generated for procedure calls? (5) **(Ref.Qn.No.19, Pg.no.70)**

MAY 2012(ARREAR)

1. a) Write an algorithm for constructing LR parser table. **(Ref.Qn.No.8, Pg.no.32)**

   b) Write the quadruple representation for the assignment statement a: =-b*(c+d) **(Ref.Qn.No.14, Pg.no.52)**

   (OR)
2. Discuss the Role of the parser. **(Ref.Qn.No.1, Pg.no.14)**

NOV 2012(REGULAR)

1. a) Write an algorithm for constructing LR parser table. **(Ref.Qn.No.8, Pg.no.32)**

   b) Consider the following grammar to construct the LR parsing table **(Ref.Qn.No.9, Pg.no.35)**

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

(OR)

2. List out and discuss the different type of intermediate code. **(Ref.Qn.No.10, Pg.no.43)**

MAY 2013(ARREAR)

1. Give the following CFG grammar G=({S,A,B},S,(a,b,x),P) with P:

$S \rightarrow A$

$S \rightarrow xb$

$A \rightarrow aAb$

$A \rightarrow B$

$B \rightarrow x$

For this grammar answer the following questions:

Compete the set of LR (1) items for this grammar. Augment the grammar with the default initial production $S' \rightarrow S\$$ as the production (0) and Construct the corresponding LR parsing table.

(OR)

2. State and write the semantic rules for Boolean expressions. **(Ref.Qn.No.17, Pg.no.61)**

NOV 2013 (REGULAR)

1. (a) Write the steps in writing a grammar for a programming language. (5) **(Ref.Qn.No.3, Pg.no.19)**

(b) Briefly write on Parsing techniques. Explain with illustration the designing of a Predictive Parser. (6)

(Ref.Qn.No.4, Pg.no.23)

(OR)

2. Describe in detail the declarations in a procedure and the methods to keep track of scope information.

(Ref.Qn.No.15, Pg.no.53)

UNIT V

Basic Optimization

Basic Optimization: Constant-Expression Evaluation – Algebraic Simplifications and Re association– Copy Propagation – Common Sub-expression Elimination – Loop-Invariant Code Motion – Induction Variable Optimization.

Code Generation: Issues in the Design of Code Generator – The Target Machine – Runtime Storage management – Next-use Information – A simple Code Generator – DAG Representation of Basic Blocks – Peephole Optimization – Generating Code from DAGs

2 MARKS

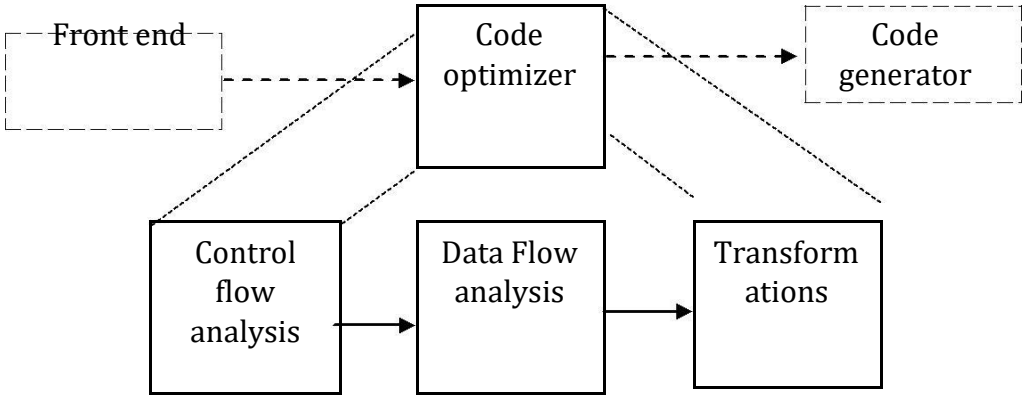
1. What is code optimization?

Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation. The techniques consist of detecting patterns in the program and replacing these patterns by equivalent and more efficient constructs. This improvements is achieved by program transformation are called ***optimization.***

2. What is optimizing compilers?

Compilers that apply code-improving transformations are called ***optimizing compilers.***

3. Give the block diagram of organization of code optimizer.



4. What are the properties of optimizing compilers?

- Transformation must preserve the meaning of programs.
- Transformation must, on the average, speed up the programs by a measurable amount.
- A Transformation must be worth the effort.

5. What are the advantages of the organization of code optimizer?

- The operations needed to implement high level constructs are made explicit in the intermediate code, so it is possible to optimize them.
- The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine

6. What are the 3 areas of code optimization?

- Local optimization
- Loop optimization
- Data flow analysis

7. Define local optimization.

The optimization performed within a block of code is called a local optimization.

8. Define constant folding.

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
- Constant folding is nothing but replacing the run-time compilation by the compile time compilation. This is done generally for the constants.
- Example: a:=(22/7) * (r*r)

9. What is propagation?

- Propagation means propagating an entity from one statement to another statement. This is done for constants.
- Evaluation or replace a variable with constant which has been assigned to it earlier.

10. Define Local transformation & Global Transformation.

- 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 local and global levels.
- Local transformations are usually performed first.

11. Give the criteria for code-improving transformations. (NOV 2011)

- Common sub expression elimination
- Copy propagation
- Dead – code elimination
- Constant folding

12. What is meant by Common Sub expressions?

An occurrence of an expression E is called a ***common sub expression***, if E was previously computed, and the values of variables in E have not changed since the previous computation.

13. What is copy propagation?

The assignment of the form *f := g* called copy statements or copies.

14. What is meant by Dead Code?

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. The statement that computes values that never get used is known ***Dead code*** or ***useless code.***

15. What are the techniques used for loop optimization?

Three techniques are important for loop optimizations are

i) Code motion

ii) Induction variable elimination

iii) Reduction in strength

16. What is code motion?

- ***Code motion***, which moves code outside a loop.
- *Code motion* is an important modification that decreases the amount of code in a loop.
- 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.

17. Define Induction variable? (MAY 2013)

The values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called ***induction variables.***

18. What is meant by Reduction in strength? (MAY 2012)

Reduction in strength, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

19. What is meant by loop invariant computation?

The transformation takes an expression that yields the same result independent of the number of times the loop is executed is known as loop invariant computation.

20. What is code generation?

- The final phase in our compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

21. What are the issues in the design of a code generator?

The various issues in design of code generator are

1. Input to the Code Generator

2. Target Programs

3. Memory Management

4. Instruction Selection

5. Register Allocation and

6. Choice of Evaluation Order

22. What are the outputs of code generator?

The output of the code generator is the target program. The output may take on a variety of forms:

1. absolute machine language,
2. relocatable machine language, or
3. assembly language.

23. What is register allocation?

- Instructions involving register operands are usually shorter and faster than those involving operands in memory. Efficient utilization of register is particularly important in generating good code.
- The use of registers are
  1. Register allocation
  2. Register assignment

24. What are the types of address mode?

The types of address modes in assembly-language

- Absolute
- Register
- Indexed
- Indirect register
- Indirect indexed

25. What is meant by activation record?

Information needed during an execution of a procedure is kept in a block of storage called activation record; storage for names local to the procedure also appears in the activation record.

26. What are the two standard storage allocation strategies?

The two standard allocation strategies are

1. Static allocation
2. Stack allocation

27. Define static and stack allocation.

- In *static allocation,* the position of an activation record in memory is fixed at compile time.
- In *stack allocation,* a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

28. What are the fields in activation records?

The activation record for a procedure has fields as

- to hold parameters,
- results
- machine status information,
- local data,
- temporary variables

29.What are the limitations of using static allocation?

- The size of a data object and constraints on its position in memory must be known at compile time.
- Recursive procedure are restricted, because all activations of a procedure use the same bindings for local name
- Data structures cannot be created dynamically since there is no mechanism for storage allocation at run time

30.When static allocation can become stack allocation? (NOV 2011)

- Static allocation can become stack allocation by using relative addresses for storage in activation records.
- The position of the activation record for the procedure is not known until run time.
- In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register.

31.What is a basic block? What are the entry points and how do you call the entry instructions?

(MAY 2013)

A ***basic block*** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

32.What are descriptors in code generation algorithm?

- The code-generation algorithm uses descriptors to keep track of register contents and addresses for names.
  1. Register descriptors
  2. Address descriptors

33.What is Register Descriptors?

- A register descriptor keeps track of what is currently in each register.
- Initially all the registers are empty.
- Each register will hold the value of zero or more names at any given time.

34.What is Address Descriptors?

- Address descriptors keeps track of location where current value of the name can be found at runtime.
- The location might be a register, a stack location or a memory address.
- This information can be stored in the symbol table and is used to determine the accessing method for a name.

35.What is DAG? (NOV 2012, 2013)

- Directed acyclic graphs (DAG) are useful data structure for implementing transformations on basic blocks.
- A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.

36.How DAG is constructing?

DAG is constructing from three-address statements is a good way of

- Determining the common sub-expressions
- Determining which names are used inside the block but evaluated outside the block and
- Determining which statements of the block could have their computed value outside the block.

37.Mention the applications of DAG?

- To automatically detect a common sub expressions.
- To determine which identifiers have their values used in the block.
- To determine which statements compute values that could be used outside the block.

38.Define peephole optimization.

The technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence whenever possible.

39.List the characteristics of peephole optimization.

The characteristics of peephole optimization are

- Redundant instruction elimination
- Unreachable Code
- Flow of control optimizations
- Algebraic simplification
- Reduction in Strength
- Use of machine idioms

40. Construct a 3-address code for (B+A) * (Y-(B+A)). (NOV 2013)

The 3-address code for (B+A) * (Y-(B+A)) t1 := B + A;

t2 := Y - t1; t3 := t1 * t2;

41.Define Flooding? (MAY 2012)

- A **flooding algorithm** is an algorithm for distributing material to every part of a graph. The name derives from the concept of inundation by a flood.
- Flooding algorithms are used in computer networking and graphics. Flooding algorithms are also useful for solving many mathematical problems, including maze problems and many problems in graph theory.

42.What is translation of symbol? (NOV 2012)

- A translation of symbols mainly a constant folding and constant propagation.
- A context-free grammar with semantic actions embedded within the right sides of the productions.

11 MARKS

1. Write a short note on Constant, Expression Evaluation? (6 marks)

Constant:

The code improving transformation as

- Constant folding
- Constant propagation

(i) Constant Folding:

- Constant - expressions evaluation or ***constant folding*** refers to the evaluation at compile time of expressions whose operands are known to be constant.
- Evaluation of an expression with constant operands to replace the expression with single value.
- This is actually compile - time evaluation.
- It makes the possible for the computations performed during the compile time itself, and thus avoids the computation during the execution time.
- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
- Constant folding is nothing but replacing the run-time compilation by the compile time compilation. This is done generally for the constants.

Example: a :=(22/7) * (r * r) → a := 3.14286 * (r * r)

- The value (22/7) can be computed during the compilation itself than computing it in each execution.

Example: i = 320 * 200 * 32

- Most compilers will substitute the computed value at compile time.

(ii) Constant Propagation:

- ***Constant propagation*** means propagating an entity from one statement to another statement. This is done for constants.
- Evaluation or replace a variable with constant which has been assigned to it earlier.
- Constant propagation is particularly important when procedures or macros are passed constant parameters.
- Constant propagation is nothing but replacement of a variable by a constant that appears on the

  right hand side of an assignment for that variable.

Example: a: =2;

Consider the three-address

statements

temp1:=4;

……

temp2:=temp1*2;

Here, the variable temp1 is propagated. This can be optimized during the compile

time itself. temp1:=4;

……

temp2:=4*2;

This is actually ***constant propagation.***
The variable should not be redefined along the path of its use.

Example:

pi: =3.14286

Area: = pi * r ** 2; → area: = 3.14286 * r**
2; This process is done during the compile time.

(iii) Expression Evaluation:

- ***Expressions evaluation*** or constant folding refers to the evaluation at compile time of expressions whose operands are known to be constant.
- Determine that all operands in an expression are constant value.
- Perform the evaluation of the expression at compile time.
- Replace the expression by its value.
- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
- The definition of the variables involved should not change.

Example:

a := b * c;

…

….

…

….

x := b * c + 5;

To generate three-address code for the above statements

temp1 := b *

c; a :=

temp1;

…….

x := temp1 + 5;

2. Write a short note on algebraic Simplifications and Re-association? (6 marks)

(i) Algebraic Simplifications:

- Algebraic simplification uses algebraic properties of operators or particular operand combinations to simplify expressions.

Expressions simplification:

- i +0 = 0 + i = i - 0 = i
- i ^ 2 = i * i (also strength reduction)
- i*5 can be done by t := i shl 3; t=t-i
- Associativity and distributive can be applied to improve parallelism (reduce the height of expression trees).
- Algebraic simplifications for floating point operations are seldom applied.
- The reason is that floating point numbers do not have the same algebraic properties as real numbers.

Examples:

1. ***Algebraic simplification using the rules***

A*1 := A

A*0 := 0

A-0 := A

A/1 := A single instruction with a constant operand A*2 := A+A

A^2 := A*A

A**2 := A*A

2. ***Initial code*** ***Algebraic simplification***

A := X**2;		A := X*X;
B := 3;		B := 3;
C := X;		C := X;
D := C*C;	⟺	D := C*C;
E := B*2;		E := B+B;
F := A+D;		F := A+D;
G := E*F;		G := E*F;

3. Algebraic transformation can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

b && true := b

b && false :=

false b || true :=

true b || false :=

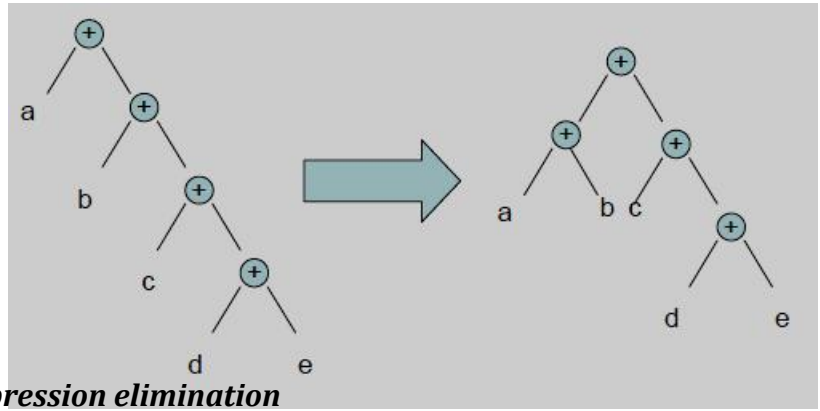
b

X * 4 :=

X<<2 16* X

:= X<<4

4. The goal is to reduce height of expression tree to reduce execution time in a parallel environment.



5. ***Common sub-expression elimination***

Transform the program so that the value of a (usually scalar) expression is saved to avoid having to compute the same expression later in the program.

For example:

x = e^3+1

…

y= e^3

is replaced (assuming that e is not reassigned in …)

with t=e^3

x = t+1

…

y=t

6. Copy propagation

- Eliminates unnecessary copy operations.

For example:

x = y

<other

instructions> t = x

+ 1

Is replaced (assuming that neither x nor y are reassigned in …) with

<other

instructions> t = y

+ 1

- Copy propagation is useful *after* common sub-expression elimination.

For example:

x = a+b

…

y = a+b

Is replaced by common sub-expression elimination into the

following code t = a+b

x = t

…

z = x

y = a+b

Here x=t can be eliminated by copy propagation.

(ii) Algebraic Re-association:

- Re-association refers to using associativity, commutativity, and distributivity to divide an expression into parts that are constant, loop invariant and variable.
- The optimization that can remove useless instructions entirely via algebraic identities.

Example:

Consider the assignment statement **b=5+a+10**

The three address code for the above sequence

```
temp1=5;                    temp1=5;                    temp1=15+
temp2=temp1+a;    ⟹        temp2=temp1+a;    ⟹        a;
temp3=temp2+10;             b=temp1;                    b=temp1;
b=temp3;
```

3. Explain principal sources of optimization or Local optimization techniques. (11 marks) (NOV 2012 MAY 2013)

- 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 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.

The examples of function preserving transformations are

1. Common sub-expression elimination
2. Copy propagation
3. Dead code elimination
4. Constant folding

The DAG representation of basic blocks showed how local common sub-expressions could be removed as the DAG for the basic block is constructed.

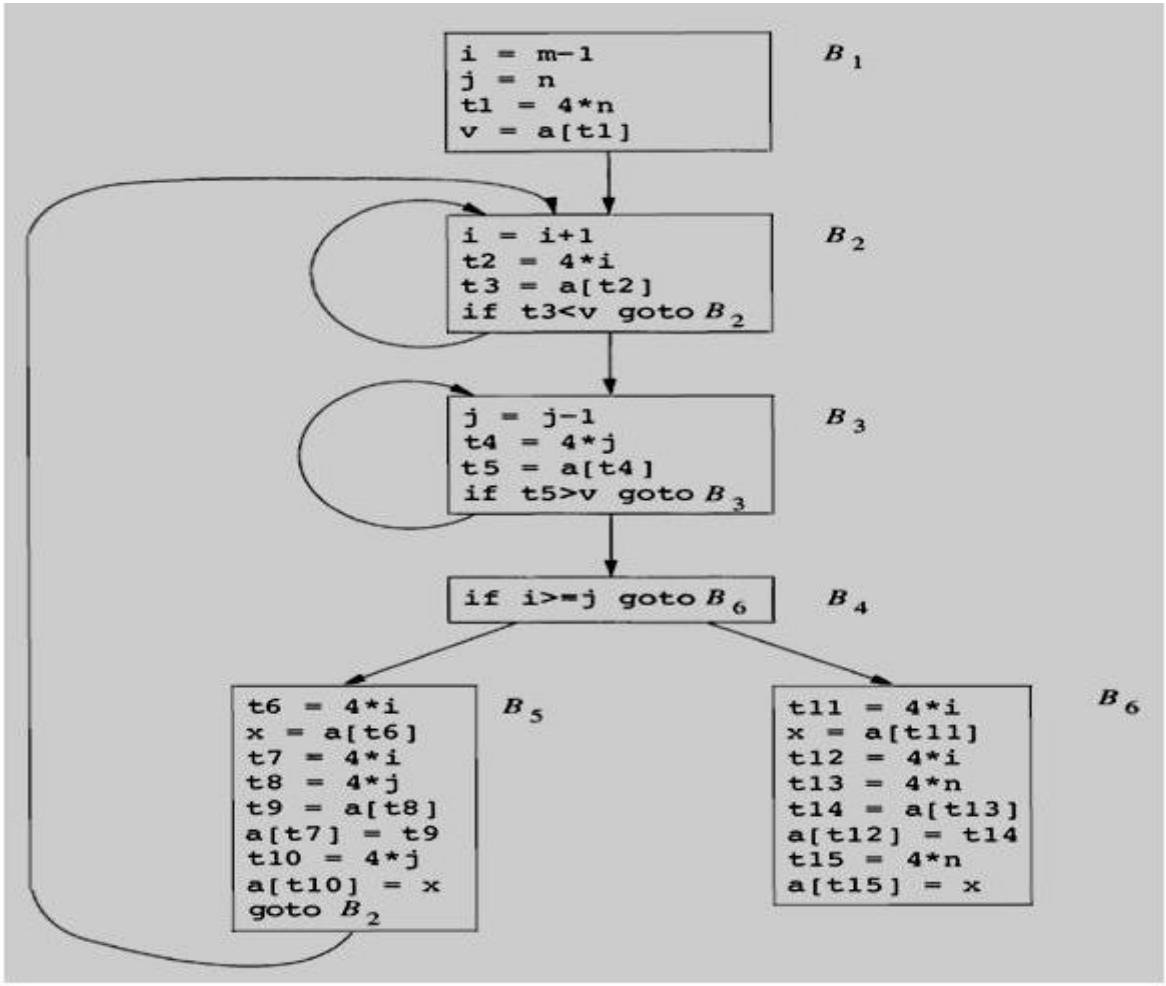
Quick sort for c program:

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Three-address code:

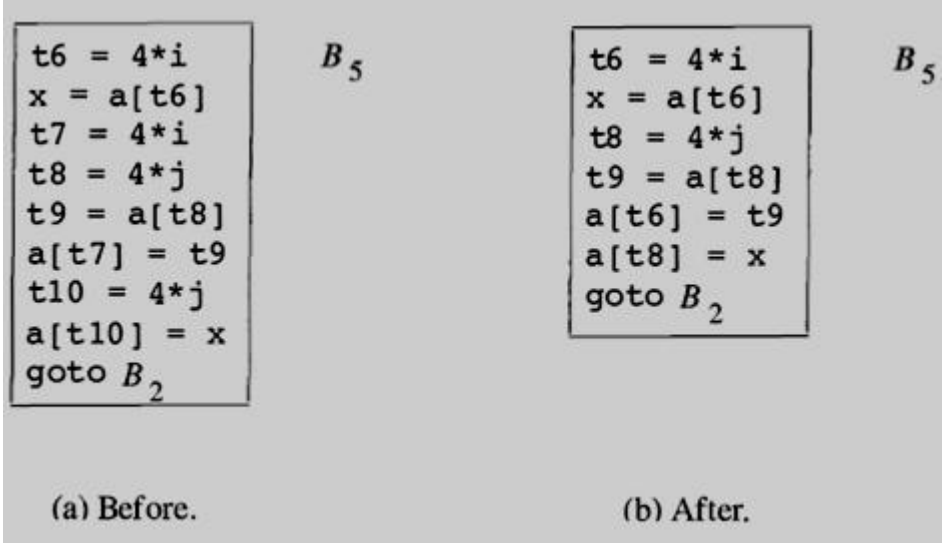
1	i = m - 1	16	$t_7 = 4 * i$
2	j = n	17	$t_8 = 4 * j$
3	$t_1 = 4 * n$	18	$t_9 = a[t_8]$
4	$v = a[t_1]$	19	$a[t_7] = t_9$
5	i = i + 1	20	$t_{10} = 4 * j$
6	$t_2 = 4 * i$	21	$a[t_{10}] = x$
7	$t_3 = a[t_2]$	22	goto (5)
8	if t_3 < v goto (5)	23	$t_{11} = 4 * i$
9	j = j – 1	24	$x = a[t_{11}]$
10	$t_4 = 4 * j$	25	$t_{12} = 4 * i$
11	$t_5 = a[t_4]$	26	$t_{13} = 4 * n$
12	if t_5 > v goto (9)	27	$t_{14} = a[t_{13}]$
13	if i >= j goto (23)	28	$a[t_{12}] = t_{14}$
14	$t_6 = 4 * i$	29	$t_{15} = 4 * n$
15	$x = a[t_6]$	30	$a[t_{15}] = x$

Flow graph:



(i) Common Sub-expression Elimination:

- An occurrence of an expression E is called a ***common sub expression***, if E was previously computed, and the values of variables in E have not changed since the previous computation.
- DAG representations of basic blocks show how local common sub-expressions could be removed as the DAG for basic block.
- A program can be calculated the same value such as an offset in an array.
- Ex: recalculates ***4*i*** and ***4*j.***
- For example, the assignments to ***t7*** and ***t10*** have the common sub-expressions ***4*i*** and ***4*j,*** respectively. They have been eliminated by using ***t6*** instead of ***t7*** and ***t8*** instead of ***t10***.



Local common sub-expression elimination

Elimination of global and local common Sub-expression:

- The result of eliminating both global and local common sub-expressions from blocks B5 and B6 in the flow graph.
- After local common sub-expressions are eliminated B5 still evaluates 4*i and 4*j.
- Both are common sub-expressions; in particular, the three statements

  t8:= 4*j; t9:= a [t8]; a [t8]:= x in B5 can be

  replaced by

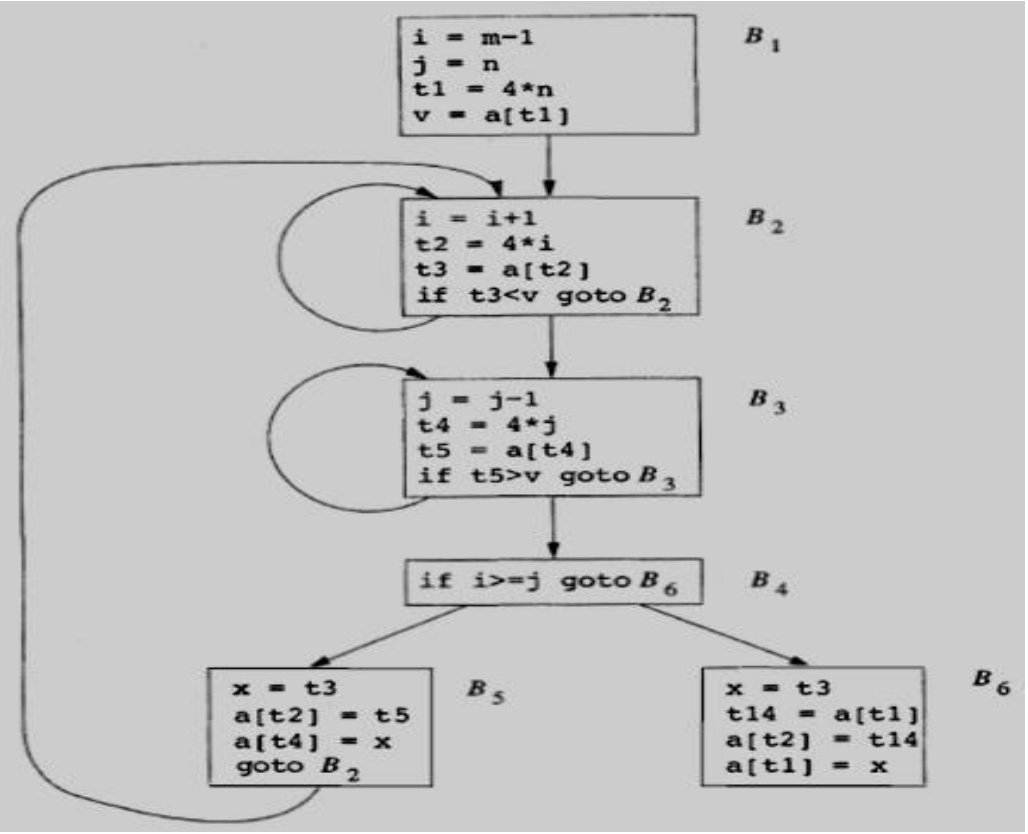
  t9:= a [t4]; a [t4]:= x using t4 computed in block B3.
- The control passes from the evaluation of 4*j in B3 to B5, there is no change in j, so t4 can be used if 4*j is needed.
- Another common sub-expression comes to light in B5 after ***t4 replaces t8***.
- The new expression a[t4] corresponds to the value of a[j] at the source level.

The statement

t9:= a [t4]; a [t6]:= t9 in B5

can therefore be replaced by

a [t6]:= t5



Elimination of global and local common Subexpression

(ii) Copy Propagation:

- Block B5 can be further improved by eliminating x using two new transformations.
- The assignments of the form **f := g** called **copy statements**, or ***copies***
- When the common sub-expression in ***c: = d+e*** is eliminated 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: = t3*** in block B5 is a copy. Copy propagation applied to B5 yields:

x := t3

a

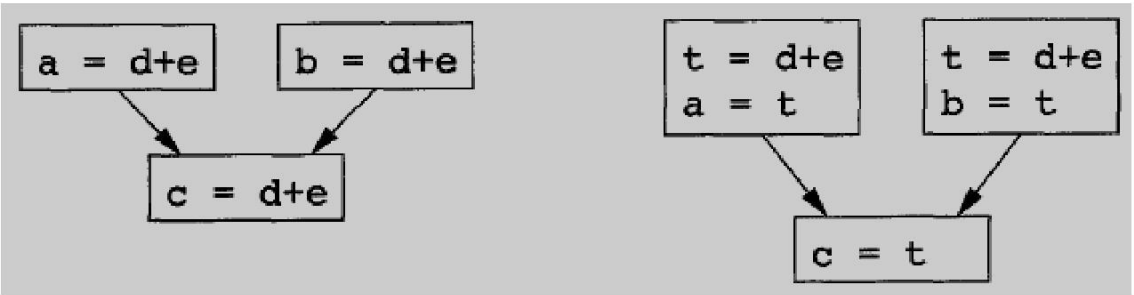
[t2]:=t

5 a

[t4]:=t

3 goto

B2



Copies introduces during common sub-expression elimination

(iii) Dead-Code Elimination:

- A variable is live at a point in a program if its value is used subsequently;
- Otherwise it is dead at that point.
- Dead or useless code statements that compute values that never get used.
- Dead Code are portion of the program which will not be executed in any path of the program can be removed.
- For example, the use of debug that is set to true or false at various points in the program, and used in

  statements like

  if (debug) print ...

- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is ***false***.
- Usually, it is because there is one particular statement

  debug :=false

- If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We can eliminate both the test and printing from the object code.

(iv) Constant folding:

- More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
- 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 [t2] :=

  t5 a [t4]

  := t3

  goto B2

4. Explain Loop optimization techniques. (11 marks) (NOV 2011, 2013) (MAY 2012)

- The 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

1. ***Code motion*** → which moves code outside a loop;
2. ***Induction-variable elimination*** →which we apply to eliminate i and j from the inner loops B2 and B3.
3. ***Reduction in strength*** → which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

(i) 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.
- 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

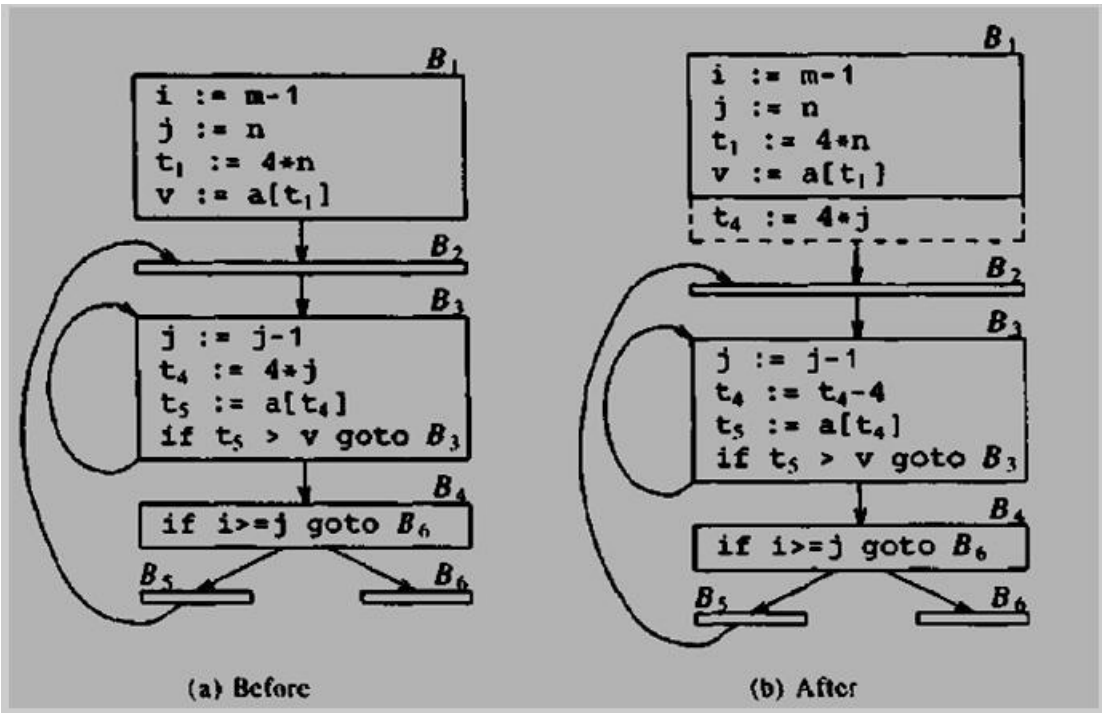
equivalent of ***t= limit-2;***

while (i<=t) /* statement does not change limit or t */

(ii) Induction Variables:

- The values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4.Such identifiers are called ***induction variables***.
- When there are two or more induction variables in a loop, by the process of induction-variable elimination.
- For the inner loop around B3, t4 is used in B3 and j in B4.

Strength reduction applied to 4*j in block B3

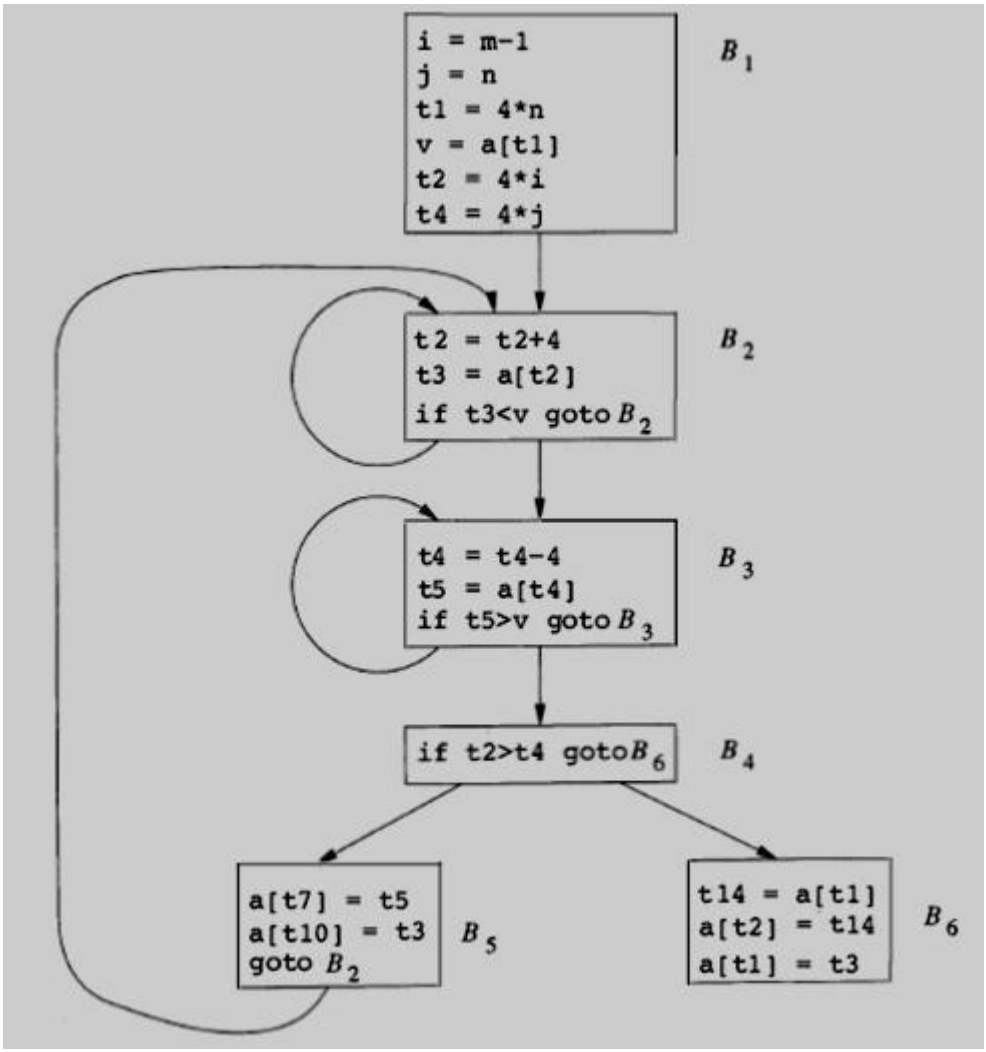


- In block B3, t4 := 4 * j holds assignment to t4 and t4 is not changed elsewhere in the inner loop around B3.
- It follows that the statement j := j - 1 the relationship t4 := 4 *j - 4 must hold.
- The replace the assignment **t4 := 4 * j** by **t4 := t4 - 4.**
- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction.

(iii) Reduction in Strength:

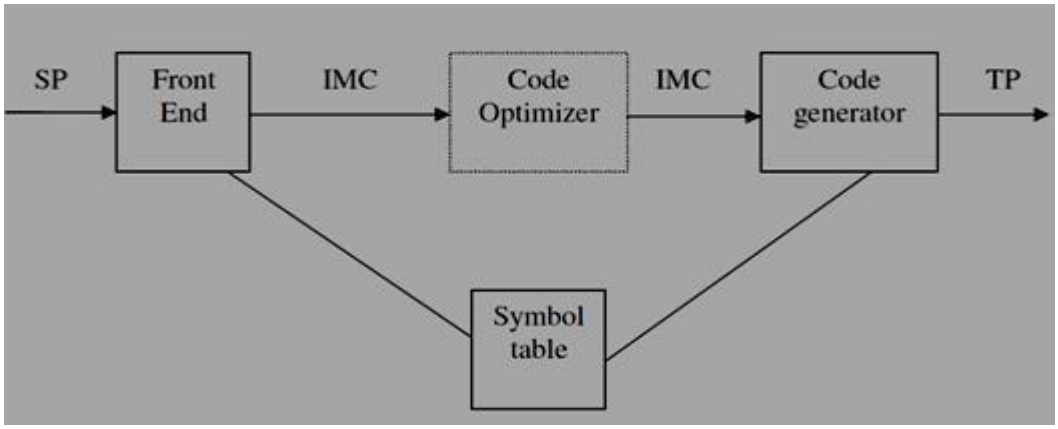
- **Reduction in strength,** which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.
- After reduction in strength is applied to the inner loop around B2 and B3, the only use in i and j to determine the outcome of test in block B4.
- The value of **i** and **t2 = 4 * i** and **j** for **t4 = 4 * j** , so the test **t2 >= t4** is equivalent to i >= j.
- Replacement i in block B2 and j in Block B3 become dead variables and assignment to the blocks, a dead code that can be eliminated.

Flow graph for induction variable elimination:



5. Explain the issues in design of a code generator? (11 marks) (NOV 2013)

- The final phase in our compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.



Issues in the design of a code generator:

While the details are dependent on the target language and the operating system, issues such as

1. Input to the code generator
2. Target Programs
3. memory management
4. instruction selection
5. register allocation and
6. evaluation order

(i)Input to the Code Generator:

- The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the

  run-time addresses of the data objects denoted by the names in the intermediate representation.
- The intermediate language, including:
  - ***linear*** representations such as ***postfix notation,***
  - ***three address*** representations such as ***quadruples,***
  - ***virtual*** representations such as ***stack machine code.***
  - ***graphical*** representations such as ***syntax trees and dags.***
- The code generation the front end has scanned, parsed, and translated the source program into a intermediate representation, the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.).

(ii) Target Programs:

- The output of the code generator is the target program.
- The intermediate code, this output may take on a variety of forms:
    1. absolute machine language,
    2. relocatable machine language, or
    3. assembly language

Absolute machine language

- Producing an absolute machine language program as output that it can be placed in a location in memory and immediately executed.
- A small program can be compiled and executed quickly.
- A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Relocatable machine language

- Producing a relocatable machine language program as output allows subprograms to be compiled separately.
- A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Assembly language

- Producing an assembly language program as output makes the process of code generation somewhat easier.
- We can generate symbolic instructions and use the macro facilities of the assembler to help generate code

(iii) Memory Management:

- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
- We assume that a name in a three-address statement refers to a symbol table entry for the name.
- Symbol-table entries are created as the declarations in a procedure.
- The type of declaration determines the width, the amount of storage, needed for the declared name.
- The symbol-table information, needed for the determined for the name in a data area for the procedure.
- The static and stack allocation of data areas , and show how names in the intermediate representation can be converted into addresses in the target code.

(iv) Instruction Selection:

- The instruction set of the target machine determines the difficulty of instruction selection.
- The instructions set are important factors

    1. ***uniformity***
    2. ***completeness***
    3. ***Instruction speeds***
    4. ***and machine idioms***

- If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.
- The efficiency of the target program, instruction selection is straightforward.
- Three- address statement can design a code skeleton that the target code to be generated.
- For example, three address statement of the form **x := y + z,** where x, y, and z are statically allocated, can be translated into the code sequence

```
MOV y, R0      /* load y into register R0 */
ADD z, R0      /* add z to R0 */
MOV R0, x      /* store R0 into x */
```

Unfortunately, this kind of statement – by - statement code generation often produces poor code. For example, the sequence of statements

```
a := b +
c d := a
+ e
```

would be translated into

```
MOV b, R0
ADD  c, R0
MOV R0, a
MOV a, R0
ADD  e, R0
MOV R0, d
```

- Here the fourth statement is redundant, and so is the third if 'a' is not subsequently used.

- The quality of the generated code is determined by its speed and size.
- Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain.

For example if the target machine has an **"increment" instruction (INC),** then the three address statement **a := a+1** may be implemented more efficiently by the single instruction ***INC a***, sequence that loads a into a register, add one to the register, and then stores the result back into a.

```
MOV a, R0
ADD  #1, R0
MOV R0, a
```

(v) Register Allocation:

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in generating good code.
- The use of registers is often subdivided into two sub-problems:
  1. During **register allocation,** we select the set of variables that will reside in registers at a point in the program.
  2. During a **register assignment** phase, we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single register values.
- Mathematically, the problem is NP-complete.
- The problem is further complicated because the hardware or the operating system of the target machine may require that certain register usage.
- Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results.
- For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs.

The ***multiplication instruction*** is of the form

```
M  x, y
```

- where x, is the multiplicand, is the even register of an even/odd register pair.
- The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The ***division instruction*** is of the form

D x, y

- where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor.
- After division, the even register holds the remainder and the odd register the quotient.

Consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

- **Ri** stands for **register i**.
- **L**, **ST** and **A** stand for **load, store and add** respectively.

The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

t := a + b	**t := a + b**
t := t * c	**t := t + c**
t := t / d	**t := t / d**
(a)	**(b)**

Two three address code sequences

L R1, a	**L R0, a**
A R1, b	**A R0, b**
M R0, c	**A R0, c**
D R0, d	**SRDA R0, 32**
ST R1, t	**D R0, d**
	ST R1, t
(a)	**(b)**

Optimal machine code sequence

(vi) Choice of Evaluation Order:

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order is another difficult, NP-complete problem.
- Initially, to avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

6. Explain the Target machine in code generator? (6 marks)

- Familiarity with the target machine and its instruction set is for designing a good code generator.
- Our target computer is a byte addressable machine with four bytes to a word and n general purpose registers, R0,R1,…., Rn-1.

The has two address instruction is of the

form **op source,**

destination

in which **op** is an **op-code** and **source** and **destination** are **data fields.**
The op-codes are

MOV (move source to destination)

ADD (add source to destination)

SUB (subtract source from destination)

- The source and destination fields are not long enough to hold the memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands or addresses.
- The source and destination of an instruction are specified by combining register and memory locations with addressing modes.
- The description, **contents (a)** denotes the contents of the register or memory address represented by a.

Addressing modes together with their assembly-language forms and associated costs are as follows:

MODE	FORM	ADDRESS	ADDED COST
Absolute	**M**	M	**1**
Register	**R**	R	**0**
Indexed	**c(R)**	c + contents(R)	**1**
Indirect indexed	***R**	contents(R)	**0**
Indirect indexed	***c(R)**	contents(c + contents(R))	**1**
Literal	**#c**	c	**1**

The following instructions are

1. A memory location M or a register R represents itself when used as a source or destination. For example, the instruction

MOV R0, M

stores the contents of register R0 into memory location M.

2. An address offset c from the value in register R is written as c(R).Thus,

MOV 4(R0), M

stores the value **contents (4 + contents (R0))** into memory location M. **3.** Indirect versions of the last two modes are indicated by prefix *. Thus,

1. MOV *4(R0), M

stores the value **contents(contents(4 + contents(R0))** into memory location M. **4.** A final address mode allows the source to be a constant:

1. MOV #1, R0

loads the constant 1 into register R0.

Instruction Costs:

- The cost of an instruction is one plus cost associated with the source and destination modes.
- The cost corresponds to the length of the instruction.
- Address mode involving registers have ***cost zero***, while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.
- The most instructions, the time taken to fetch an instruction from memory exceeds the time spent executing the instruction.
- By minimizing the instruction length is to minimize the time taken to perform the instruction.

1. The instruction **MOV R0, R1** copies the contents of register R0 into register R1.***This instruction has cost one.***
2. The instruction **MOV R5, M** copies the contents of register R5 into memory location M. ***This instruction has cost two.***
3. The instruction **ADD #1, R3** adds the constant 1 to the contents of register 3, and ***cost has two.***
4. The instruction **SUB 4(R0), *12(r1)** stores the value

contents (contents (12 + (contents (R1)) – contents (contents (4+R0)) into the destination ***12(r1).** ***The cost of the instruction is three.***

- MOV R0, R1 → **cost = 1**
- MOV *R0,*R1 → **cost = 1**
- MOV R0, a → **cost = 2**
- MOV a, R0 → **cost = 2**
- MOV #1, R0 → **cost = 2**
- MOV a, b → **cost = 3**

The three- address statement of the form a**: = b + c,** where b and c are simple variables in distinct memory location denoted by these names.

1. MOV b, R0
   ADD c, R0
   MOV R0, a
   cost = 6

2. MOV b, a
   ADD c, a
   cost = 6

Assume R0 = a, R1 = b, R2 = c, respectively

3. MOV *R1,*R0
   ADD *R2,*R0
   cost = 2

Assume R1 = b, R2 = c

4. ADD R1, R2
   MOV R1, a
   cost = 3

7. Explain the run-time storage management? (11 marks)

- The semantic of procedures in a language determines how names are bound to storage during execution.
- Information needed during an execution of a procedure is kept in a block of storage called ***activation record;*** storage for names local to the procedure also appears in the activation record.

The two standard allocation strategies are

1. ***Static allocation***
2. ***Stack allocation***

- In ***static allocation,*** the position of an activation record in memory is fixed at compile time.
- In ***stack allocation,*** a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

The activation record for a procedure has fields as

- to hold parameters,
- results
- machine status information,
- local data,
- temporary variables.

The run time allocation and de-allocation of activation record occurs as part of the

- procedure calls and
- return sequences

The three- address statement as

1. call
2. return
3. halt and
4. action

(i) Static Allocation:

- Consider the code needed to implement static allocation.
- A **call** statement in the intermediate code is implemented by the sequence of two target-machine instructions.
- A ***MOV*** instruction to save the return address and a ***GOTO*** transfers control to the target code for the

  called procedure:

  MOV #here+20, callee.static_area

  GOTO callee.code_area

Where

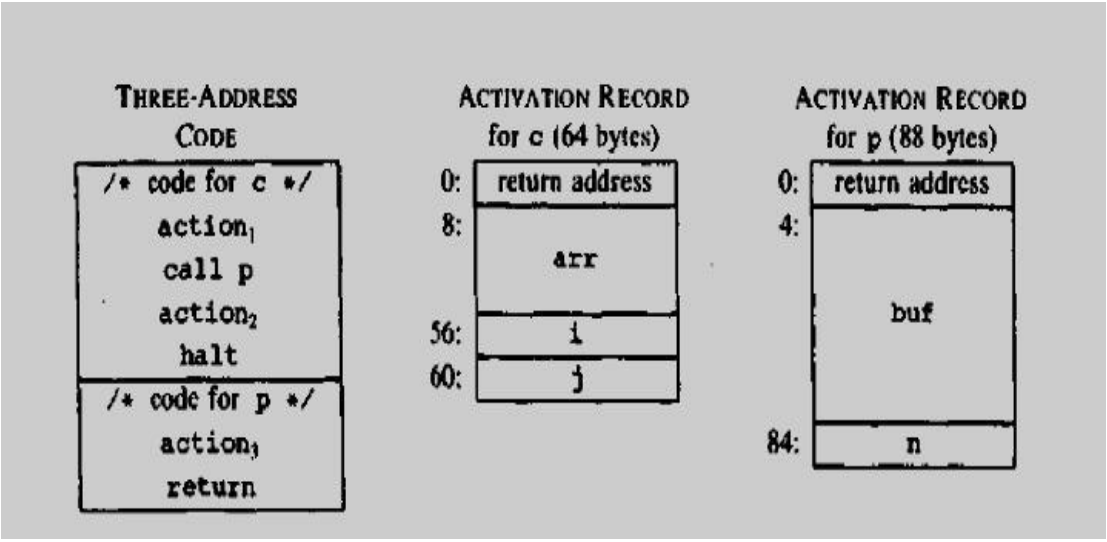
- The ***callee.static_area*** and ***callee.code_area*** are constants referring to the address of the activation record and the first instruction for the called procedure respectively.
- The source **#here+20** in the **MOV** instruction is the literal return address.
- The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is HALT.

Return from procedure ***callee*** is implemented by

GOTO *callee.code_area

- which transfers control to the address saved at the beginning of the activation record.

Input to code generator:



Target code for the input:

```
                                /*code for c*/

100:  ACTION1
120:  MOV  #140,364          /*save return address 140 */
132:   GOTO  200             /* call p */
140:  ACTION2
160:   HALT
        ……
                             /*code for p*/
200:   ACTION3
220:   GOTO  *364            /*return to address saved in location 364*/
        ……
                             /*300-363 hold activation record for c*/
300:                         /*return address*/
304:                         /*local data for c*/

        ……                   /*364-451 hold activation record for p*/
364:                         /*return address*/
368:                         /*local data for p*/
```

(ii) Stack Allocation:

- Static allocation can become stack allocation by using relative addresses for storage in activation records.
- The position of the record for an activation of a procedure is not known until run time.
- In ***stack allocation,*** this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register.
- Relative addresses in an activation record can be taken as offsets from any known position in the activation record.
- A register SP a pointer to the beginning of the activation record on the top of the stack.
- When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
- After control returns to the caller, it decrements SP, thereby de-allocating the activation record of the called procedure.

The code for the first procedure initializes the stack by setting SP to the start of the stack area in memory:

```
MOV #stackstart, SP          /* initialize the stack */
code for the first procedure
HALT                         /* terminate execution */
```

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD #caller.recordsize, SP
MOV #here+16, SP             /* save return address */
GOTO callee.code_area
```

- The attribute ***caller.recordsize*** represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record.
- The source ***#here+16*** in the ***MOV instruction*** is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

```
GOTO *0(SP)                  /*return to caller*/
```

The reason for using *0(SP) in the GOTO instruction is that we need two levels of indirection:

- 0(SP) is the address of the first word in the activation record and
- *0(SP) is the return address saved there.

- The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value.
- That is, after the subtraction SP points to the beginning of the activation record of the caller:

```
SUB #caller.recordsize, SP
```

Target code for stack allocation:

Three-address code

```
/*code for s*/
  action1
  call q
  action2
  halt

/*code for p*/
  action3
  return

/*code for q*/
  action4
  call p
  action5
  call q
  action6
  call q
  return
```

Three address code for stack allocation:

```
                          /*code for s*/
100: MOV  #600, SP    /*initialize the stack*/
108: ACTION1
128: ADD  #ssize, SP    /*call sequence begins*/
136: MOV  #152, *SP     /*push return address*/
144: GOTO  300          /*call q*/
152: SUB  #ssize, SP    /*restore SP*/
160: ACTION2
180: HALT
     .........
```

```
                          /*code for p*/
200: ACTION3
220: GOTO  *0(SP)         /*return*/
     .........
```

```
                              /*code for q*/
300: ACTION4                  /*conditional jump to  456*/
320: ADD  #qsize, SP
328: MOV  #344, *SP           /*push return address*/
336: GOTO  200                /*call p*/
344: SUB  #qsize, SP
352: ACTION5
372: ADD  #qsize, SP
380: MOV  #396, *SP           /*push return address*/
388: GOTO  300                /*call q*/
396: SUB  #qsize, SP
404: ACTION6
424: ADD  #qsize, SP
432: MOV  #448, *SP           /*push return address*/
440: GOTO  300                /*call q*/
448: SUB #qsize, SP
456: GOTO  *0(SP)             /*return*/
       .........
600:                          /*stack starts here*/
```

Run time addresses for names:

- The storage allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed.
- Assume that a name in a three-address statement is really a pointer to a symbol-table entry for the name; it makes the compiler more portable, since the front end need not be changed even if the compiler is moved to a different machine where a different run-time organization is needed.
- Names must be replaced by code to access storage locations. Consider the simple three-address statement x: = 0.
- After the declarations in a procedure are processed, suppose the symbol-table entry for x contains a relative address 12 for x.
- First consider the case in which x is in a statically allocated area beginning at address *static.* Then the actual run-time address for x is *static*+12.
- The assignment x := 0 then translates into

  static [12] := 0
- If the static area starts at address 100, the target code for this statement is

  MOV #0, 112
- Suppose x is local to an active procedure whose display pointer is in register R3. Then we may translate the copy x := 0 into the three-address statements

  t1 := 12+R3

  ***t1 := 0**

in which t1 contains the address of x. This sequence can be implemented by the single machine instruction

MOV #0, 12 (R3)

The value in R3 cannot be determined at compile time.

8. What is Next use information? Discuss (5 marks) (MAY 2013)

- Next use information about names in basic block.
- If the name in a register is no longer needed, then the register can be assigned for the some other name.
- The keeping a name in storage only it will be used subsequently can be applied in a number of contexts.

Computing Next Uses:

- The use of a name in a three-address statement as follows:
- Suppose three-address statements i assigns a value to x.
- If statement j has x as an operand and control can flow from statement i to j along a path that has no assignments to x, then the statement j uses the value of x computed at i.
- The three address statement ***X = Y op Z*** the next uses of X, Y and Z.
- The algorithm to determine next uses makes a backward pass over each basic block.
- Assume all temporaries are dead on exit and all user variables are live on exit.

Algorithm to compute next use information:

Suppose three address statement **i : X := Y op Z** in backward scan, the following

1. Attach to statement i, information currently found in the symbol table regarding the next use and live-ness of X, Y and Z.
2. In symbol table, set X to "not live" and "no next use ".
3. In symbol table, set Y and Z to be "live" and next use of Y and Z to i.

Storage for Temporary Names:

- The two temporaries into the same location if they are not live simultaneously.
- All temporaries are defined and used within basic blocks; next-use information can be applied to pack temporaries.
- In the basic block can be packed into two locations. These locations correspond to t1 and t2:

Example: X = a * a + 2(a * b) + (b * b)

1. t1 = a * a
2. t2 = a * b
3. t3 = 2 * t2
4. t4 = t1 + t3
5. t5 = b * b
6. t6 = t4 + t5
7. X = t6

Statement	Symbol Table		
1. t_1:use(4)	t_1	Dead	Use in 4
2. t_2:use(3)	t_2	Dead	Use in 3
3. t_3:use(4), t_2 not live	t_3	Dead	Use in 4
4. t_4:use(6), t_1 t_3 not live	t_4	Dead	Use in 6
5. t_5:use(6)	t_5	Dead	Use in 6
6. t_6:use(7), t_4 t_5 not live	t_6	Dead	Use in 7
7. no temporary is live			

9. Explain the four issues in the design of a simple code generator. Generate the code for a simple statement. (NOV 2013) (11 marks)

- The code-generation generates target code for a sequence of three address statements.
- Each three-address statement operands are currently stored in register.
- Assume that computed results can be left in registers as long as possible, storing them only
  - **(i)** If their register is needed for another computation
  - **(ii)** Just before a procedure call, Jump or labeled statement.

Register and Address Descriptors:

The code-generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. Register descriptors
2. Address descriptors

Register Descriptors:

- A register descriptor keeps track of what is currently in each register.
- Initially all the registers are empty.
- We assume that initially the register descriptor shows that all register are empty.
- The code generator for the block progresses, each register will hold the value of zero or more names at any given time.

Address Descriptors:

- Address descriptors keeps track of location where current value of the name can be found at runtime.
- The location might be a register, a stack location or a memory address.
- This information can be stored in the symbol table and is used to determine the accessing method for a name.

Code- Generation Algorithm:

The code generation algorithm takes as input a sequence of three address statement constituting a basic block.

The three address statement of the form **X = Y op Z**

STEP 1:

- Invoke a function **getreg ()** to determine the location L, where the result of computation Y op Z should be stored.
- L will usually be a register or memory location.

STEP 2:

- The address descriptor for Y to determine Y', the current location of Y.
- Prefer the register for Y', if the value Y is currently both in register and memory location.
- Generate the instruction MOV Y' in L.

STEP 3:

- Generate the instruction op Z', L where Z' is a current location of Z.
- The value Z is currently both in register and memory location.
- Update address descriptor of X to indicate that X is in location L.

STEP 4:

- If the current values of Y and Z have no next use, are not live on exit from the block.
- Register descriptor to indicate after execution of X=Y op Z.

Consider the assignment statement of the form **d: = (a - b) + (a - c) + (a - c**) might be translated into three address code sequence

t := a - b

u := a -

c v := t

+ u d :=

v + u

The *getreg ()* function:

The function getreg returns the location L to hold the value of x for the assignment **x: =y *op* z.**

The algorithm for getreg:

1) If the name y is in a register, that holds the value of no other names and y is not live and has no next use after the execution of y = x *op* z, then a. return L. Update the address descriptor of y , so that y is no longer in L.
2) Failing (1), return an empty register for L if there is one.
3) Failing (2), if x has a next use in the block, or if *op* requires a register then a. find an occupied register R. MOV(R,M) if value of R is not in proper M. If R holds value of many variables, generate a MOV for each of the variables.
4) Failing (3), select the memory location of x as L.

Code Sequence:

Statement	Code Generated	Register descriptor	Address descriptor
		Registers empty	
$t_1 = a - b$	MOV a,R_0 SUB b,R_0	R_0 contains t_1	t_1 in R_0
$t_2 = a - c$	MOV a,R_1 SUB c,R_1	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
$t_3 = t_1 + t_2$	ADD R_1,R_0	R_0 contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$d = t_3 + t_2$	ADD R_1,R_0 MOV R_0,d	R_0 contains d	d in R_0 d in R_0 and memory

Conditional Statements:

- Machines implement conditional jumps in one of two ways.
- One way is to branch if the value of a designated register have six conditions: negative, zero, positive, non-negative, non-zero and non-positive.
- The machine three address statement such as **if X < Y goto Z** can be implemented by

  Subtracting Y from X in register R and

  Then jumping to Z, if the value in register R is negative.

- A second approach, common to many machines, uses a set of ***condition code*** to indicate whether last quantity computed or loaded into a location is negative, zero, or positive.
- ***Compare instruction (CMP)*** sets the codes without actually computing the value.
- ***CMP X, Y*** sets condition codes to positive if X > Y and so on.
- A conditional-jump machine instruction makes the jump if a designated condition <, +,>, <, ≤, ≠ or ≥.
- The instruction ***CJ<=Z*** to mean "jump to Z if the condition code is negative or zero".

For example, **if X < Y goto Z** could be implemented by

```
Cmp  X, Y
CJ <  Z
```

The condition code descriptor

```
X := Y + Z
if X < 0 goto L
```

by

```
MOV Y, R0
ADD  Z, R0
MOV R0,
X CJ < L
```

The condition code is determined by x after ADD Z, R0...

10.Explain the DAG representation of basic block? (11 marks) (MAY 2013)

- ***Directed acyclic graphs (DAG)*** are useful data structure for implementing transformations on basic blocks.
- A ***dag*** gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.

Constructing a dag from three-address statements is a good way of

- Determining the common sub-expressions (expressions computed more than once) within a block.
- Determining which names are used inside the block but evaluated outside the block and
- Determining which statements of the block could have their computed value outside the block.

A ***dag for a basic block*** is a directed acyclic graph has following labels on the nodes:

1) Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to name we determine whether the L-value or R-value name is created; most leaves represent R-values. The leaves represent initial values of names and we subscript them with 0 to avoid confusion.

2) Interior nodes are labeled by an operator symbol.

3) Nodes are also optionally given a sequence of identifiers for labels. The interior nodes represent computed values and the identifiers labeling a node.

Each node of a flow graph can be represented by a dag, since each node of the flow graph stands for a basic block.

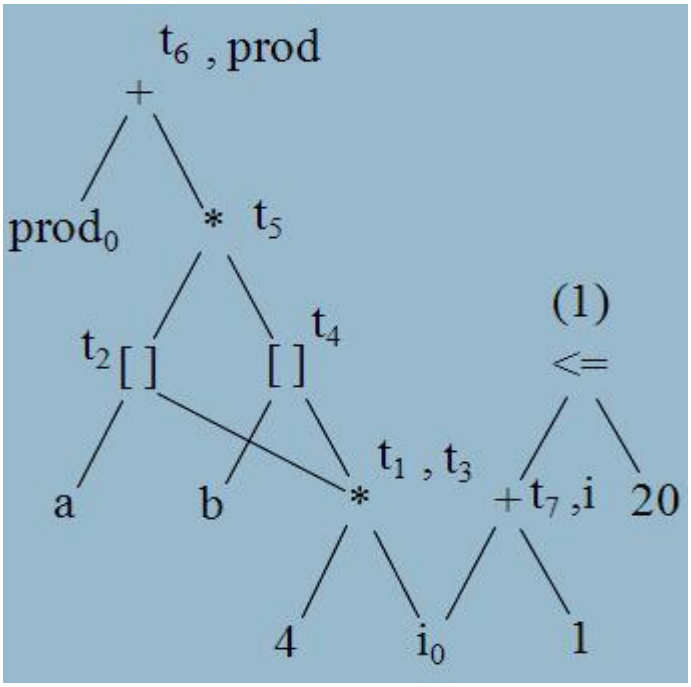
The source program

```
begin
        prod :=
        0; i :=1;
        do begin
                prod := prod + a[ i ] * b[
                i ]; i := i + 1;
        end
        while i <= 20
end
```

Three address code as

1. t1 := 4 * i
2. t2 := a[t1]
3. t3 := 4 * i
4. t4 := b[t3]
5. t5 := t2 * t4
6. t6 := prod + t5
7. prod := t6
8. t7 := i + 1
9. i := t7
10. if i <= 20 goto (1)

DAG Representation:



Dag Construction:

Input: a basic block.

Output: a dag for the basic block containing the following information:

1. A ***label*** for each node. For leaves the label is an identifier (constants permitted) and for interior nodes an operator symbol.
2. For each node a (possibly empty) list of attached identifiers (constants not permitted).

Method: Initially assume there are no nodes, ***node*** is undefined for all arguments. Suppose the and three address statements in either **(i) x:=** current **(ii) x:= op y (iii) x:= y.** A relational operator **y op z** goto case (i), with X undefined. like if i<=20

(1) If ***node(y)*** is undefined, created a leaf labeled y, let ***node(y)*** be this node. In case (i) if ***node (z)*** is undefined, create a leaf labeled z and that leaf be ***node (z).***

(2) In case (i) determine if there is a node labeled ***op,*** whose left child is ***node(y)*** and right child is ***node(z).*** If not create such a node, let be n. case (ii), (iii) similar.

(3) Delete x from the list attached identifiers for ***node(x).*** Append x to the list of identify for node n and set ***node(x)*** to n.

Application of Dags:

The applications of DAGs are

- To automatically detect a common sub expressions.
- To determine which identifiers have their values used in the block.
- To determine which statements compute values that could be used outside the block.

11.Explain the peephole optimization. (11 marks) (NOV 2011, 2012)

- The technique for locally improving the target code is ***peephole optimization***, a method for trying to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence whenever possible.
- Peephole optimization as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.
- **Peephole** is a small, moving window on the target program.

The characteristics of peephole optimization are

- Redundant instruction elimination
- Unreachable Code
- Flow of control optimizations
- Algebraic simplification
- Reduction in Strength
- Use of machine idioms

(i) Redundant loads and stores:

Consider the instruction sequence

1) **MOV R0, a**
2) **MOV a, R0**

- We can delete instruction (2) because whenever (2) is executed.
- Instruction (1) the value of a is already in register R0.

(ii) Unreachable code:

- Peephole optimization is the removal of unreachable instructions.
- An unlabeled instructions immediately following unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.

Consider following code segments that are executed only if a variable debug is 1. In C , the source code as

```
# define debug 0
……
if (debug) {
        print debugging information
}
```

In the intermediate representation the if statement may be

translated as **if debug = 1 goto L1**

goto L2

L1: print debugging

information L2:

Peephole optimization is to eliminate jump over

jumps **if debug ≠ 1 goto L2**

print debugging information

L2:

Since debug is set to 0 at the beginning of the program, constant propagation should

replace **if 0 ≠ 1 goto L2**

print debugging information

L2:

The argument of the first statement evaluates to a constant true, it can be replaced by goto L2. Then all the statements that print debugging are unreachable and can be eliminated one at a time.

(iii) Flow of control optimizations:

- The intermediate code generation algorithms are frequently produces
  - Jumps to jumps
  - Jumps to conditional jumps or
  - Conditional jumps to jumps
- The unnecessary jumps can be eliminated either in intermediate code or target code in peephole optimization.

Replace the jump

seque

nce

goto

L1

…..

L1: goto L2

by the sequence

goto L2

…..

L1: goto L2

If there are no jumps to L1, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

Similarly, the sequence

	if a < b goto L1			if a < b goto L2
	…….	can be replaced by		…….
L1 :	goto L2		L1 :	goto L2

Finally, there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the

sequence goto L1

…….

L1: if a < b goto L1 L3:

may be replaced by

if a < b goto L2

goto L3

…….

L3:

(iv) Algebraic Simplification:

- The amount of algebraic simplification that can be attempt through peephole optimization.
- For example, statements such as

x := x + 0

or

x := x * 1

are produced by straightforward intermediate code generation algorithms and they can be eliminated easily through peephole optimization.

(v) Strength reduction:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.
- Certain machine instructions are cheaper than others and can be used as special cases of more expensive operators.
- For example
  - Replace X^2 by X * X
  - Fixed point multiplication or division by a power of two is cheaper to implement as a shift.
  - Fixed point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

(vi) Use of Machine idioms:

- The target machines have hardware instructions to implement specific operations efficiently.
- The use of instruction can reduce execution time significantly.
- For example, machines have ***auto-increment and auto –decrement addressing modes.***
- These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
- These modes can also be used in code for statements i: = i+1.

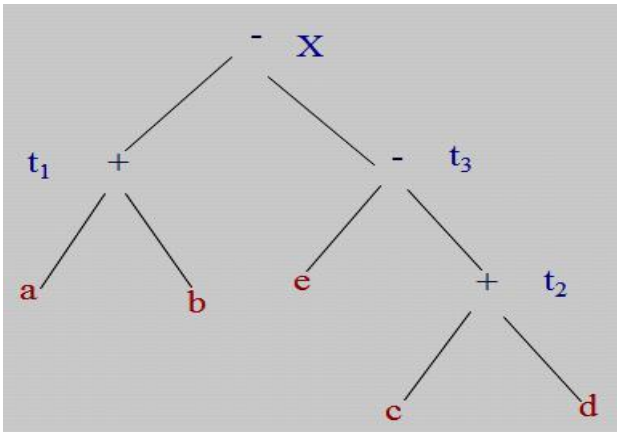
12.Write note on Generating code from DAGs. (11 marks) (MAY 2012)

- To generate code for a basic block from its DAG representation.
- Dag shows how to rearrange the order of final computation sequence from linear representation of three - address statements or quadruples.
- We can improve the program length or few no.of temporaries used.
- This algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.

(i) Rearranging the Order:

- Consider how the order in which computations are done can affect the cost of resulting object code.
- Consider the following basic block whose dag representation

t1 := a + b t2 := c + d t3 := e – t2 X := t1 – t3



Dag for basic block

- The syntax directed translation of the expression X: = (a + b) - (e - (c + d)) by the algorithm.
- Generate code for the three address statement using the algorithm

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, X
```

We rearranged the order of the statements so that the computation of t1 occurs immediatedly before that of t4 as:

```
t2 := c +
d t3 := e
–t2 t1 :=
a + b X
:= t1 –
t3
```

Using the code generation algorithm, the code sequence as

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R1, X
```

(ii) A Heuristic Ordering for Dags:

- The heuristic ordering algorithm which attempts as far as possible to make the evaluation of a node immediately follows the evaluation of its leftmost argument.
- The order of node can be edge relationship of the DAG.
- The edges are procedure calls, array or pointer assignments.

Node listing Algorithm:

```
while unlisted interior nodes remain do
    begin
        select an unlisted node n, all of whose parents have
          been listed;
        list n;
        while the leftmost child m of n has no unlisted parents
        and is not a leaf do
            /* since n was just listed, surely m is not yet listed */
            begin
                list m;
                n := m
            end
    end
```

The ordering corresponds to the sequence of three-address statements:

t8 : = d

+e t6 : = a

+ b t5 : =

t6 - c t4 :

= t5 * t8

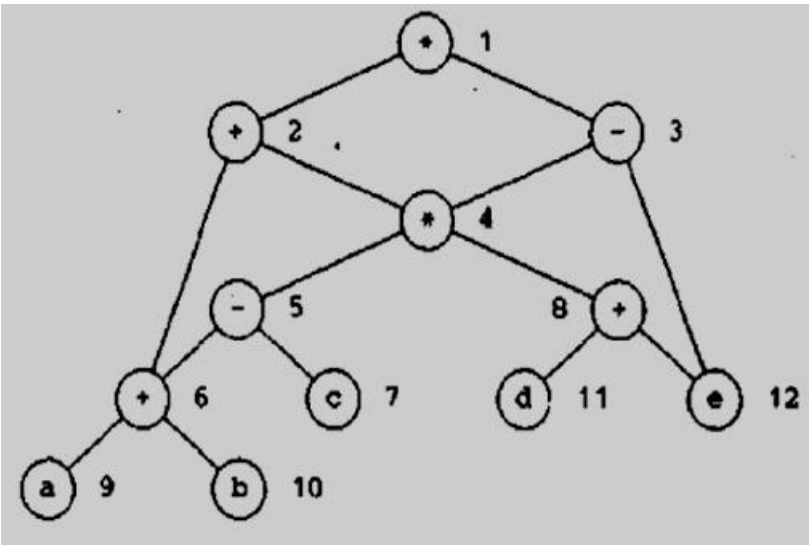
t3 : = t4 -

e t2 : = t6

+ t4 t1 : =

t2 * t3

A DAG:



(iii) Optimal Ordering for Trees:

- A simple algorithm to determine the optimal order in which to evaluate a sequence of quadruples is tree.
- Optimal ordering means the order that yields the shortest instruction sequence, over all instructions sequences that evaluate the tree.

The algorithm has two parts.

1. The first part labels each node of the tree, bottom-up, with an integer that denotes the fewest number of registers required to evaluate the tree with no stores of intermediate results.
2. The second part of the algorithm is a tree traversal whose order is governed by the computed node labels. The output code is generated during the tree traversal.

The Labeling Algorithm:

- The term "Left leaf "to mean node that is a leaf and left descendent of the parent.
- All other leaves are referred to as "right leaves".
- The labeling can be done by visiting nodes in a bottom-up order so that a node is not visited until all its children are labeled.
- The order in which parse tree nodes are created is suitable if the parse tree is used as intermediate code.
- In this case, the labels can be computed as a syntax-directed translation.

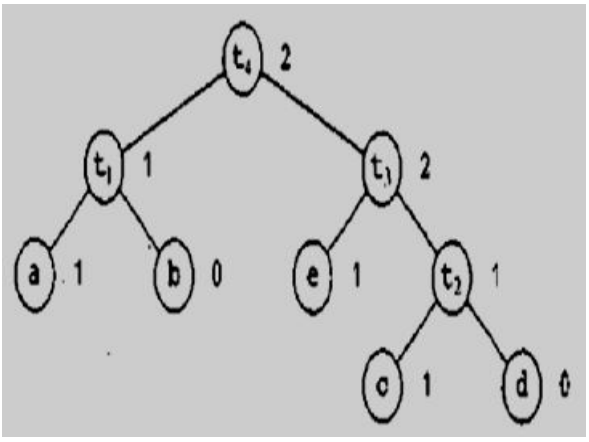
The important ***n*** is a ***binary node*** and its children have labels l1 and l2,

$$\textbf{label(n)} = \begin{cases} \textbf{max(l1, l2)} & \textbf{if l1} \neq \textbf{l2} \\ \textbf{l1+1} & \textbf{if l1=l2} \end{cases}$$

Label computations:

```
if n is a leaf then
    if n is the leftmost child of its parent then
        LABEL(n) := 1
    else LABEL(n) := 0
else /* n is an interior node */
    begin
        let n1, n2, . . . , nk be the children of n ordered by LABEL,
          so LABEL(n1) ≥ LABEL(n2) ≥ · · · ≥ LABEL(nk);
        LABEL(n) := max (LABEL(ni)+i−1)
                    1≤i≤k
    end
```

- A post-order traversal of the nodes visits the nodes in the order **a b t1 e c d t2 t3 t4.**
- ***Node a*** is labeled 1 ***left leaf.***
- ***Node b*** is labeled 0 ***right leaf.***
- Node t1 is labeled 1 because the labels of its children are unequal and the maximum label of a child is 1.



(iv) Multi-register Operations:

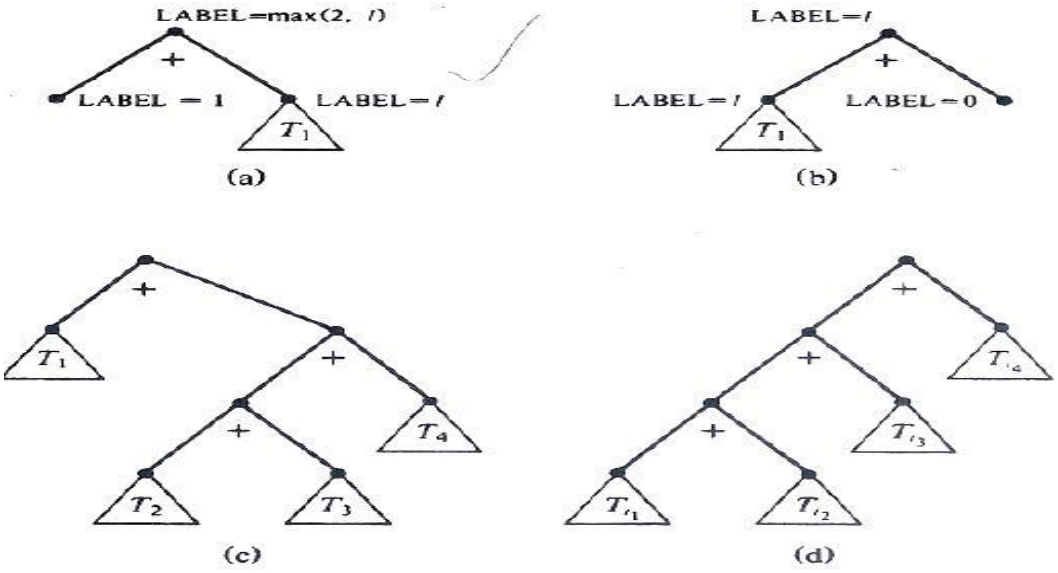
- The labeling algorithm to handle operations like multiplication, division or a function call, which normally require more than one register to perform.
- The labeling algorithm ***label (n)*** is always at least the number of registers required by the operation.
- If multiplication requires two registers, in the binary case use

$$\text{label}(n) = \begin{cases} \max(2, l1, l2) & \text{if } l1 \neq l2 \\ l1+1 & \text{if } l1 = l2 \end{cases}$$

- where l1 and l2 are the labels of the children of n.

(v) Algebraic Properties:

- The algebraic laws for various operators, the opportunity to replace a given tree T by one with smaller labels and fewer left leaves.



(vi) Common Sub-expressions:

- When there are common sub-expressions in a basic block, the corresponding dag will no longer be a tree.
- The common sub-expressions will corresponds to nodes with more than one parent called **shared nodes.**

UNIVERSITY QUESTIONS

2 MARKS

1. When static allocation can become stack allocation?**(NOV 2011) (Ref.Qn.No.30, Pg.no.6)**
2. Give the criteria for code-improving transformations. **(NOV 2011) (Ref.Qn.No.11, Pg.no.3)**
3. Define Flooding.**(MAY 2012) (Ref.Qn.No.41, Pg.no.8)**
4. What is mean by Reduction in Strength? **(MAY 2012) (Ref.Qn.No.18, Pg.no.4)**
5. Define DAG. **(NOV 2012) (NOV 2013) (Ref.Qn.No.35, Pg.no.7)**
6. What is translation of symbol? **(NOV 2012) (Ref.Qn.No.42, Pg.no.8)**
7. What is a basic block? What are the entry points and how do you call the entry instructions? **(MAY 2013)**

   (Ref.Qn.No.31, Pg.no.6)
8. Define Induction variables? **(MAY 2013) (Ref.Qn.No.17, Pg.no.4)**
9. Construct a 3-address code for (B+A) * (Y-(B+A)). **(NOV 2013) (Ref.Qn.No.40, Pg.no.8)**

11 MARKS

NOV 2011(REGULAR)

1. Describe the procedure for elimination of induction variables. **(Ref.Qn.No.4, Pg.no.19)**

   (OR)
2. Explain the peephole optimization. **(Ref.Qn.No.11, Pg.no.43)**

MAY 2012(ARREAR)

1. Write note on Generating code from DAGs. **(Ref.Qn.No.12, Pg.no.47)**

   (OR)
2. Explain Loop optimization techniques. **(Ref.Qn.No.4, Pg.no.19)**

NOV 2012(REGULAR)

1. Write note on peephole optimization. **(Ref.Qn.No.11, Pg.no.43)**

   (OR)
2. Explain Local optimization techniques. **(Ref.Qn.No.3, Pg.no.14)**

MAY 2013(ARREAR)

1. a) Explain Elimination of common sub expression during code optimization. Define for the expression

(a+b)-(a+b)/4 (6) **(Ref.Qn.No.3, Pg.no.14)**

b) What is Next use information? Discuss (5) **(Ref.Qn.No.8, Pg.no.36)**

(OR)

2. Define Directed Acyclic Graph. How is it related to Basic blocks? Construct a DAG representation for the following Basic block stating their steps. **(Ref.Qn.No.10, Pg.no.41)**

D: =B*C

E: =A+B

B: =B*C

A: =E-D.

NOV 2013 (REGULAR)

1. Explain briefly any three of the commonly used code optimization techniques. **(Ref.Qn.No.4, Pg.no.19)**

(OR)

2. Explain the four issues in the design of a simple code generator. Generate the code for a simple statement.

(Ref.Qn.No.5, Pg.no.22)