LANGUAGE TRANSLATORS

QUESTION BANK WITH ANSWERS

Note: * - Repeated Questions

UNIT - 1

Part - A

1. What is interpreter?

Interpreter is a set of programs which converts high level language program to machine language program line by line.

2. What are the fields available in an assembly language instruction?

Opcode, operand ,label and address

3. Define assembler?

The Assembler function is to translate source program into object program.

The translation of source program to object code requires accomplishing the following function.

- Converts mnemonic operation code to their machine language equivalents.
  Eg. Translate RETADR to 1033

4. Define System Software?

It 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. Define system software.

It 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, loader .

6. Define Interpreter.

Interpreter is a set of programs which converts high level language program to machine language program line by line.

7. Define instruction set.

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. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor

8. How could literals be implemented in one pass assembler

- The basic data structure needed for the assembler to handle literal operand is LITTAB.

LITTAB Contains

Address assigned to the operand when it is placed in a literal pool.
Literal name
Operand value
Length

PASS1

- As each literal operand is recognized during pass1 the assembler reaches the LITTAB specifies literal name.
- If the literal is already present no action,otherwise it is added to the LITTAB.

9. What is meant by literal pool?

All literal operands used in a program are gathered into one or more literal pools Literals are placed into the literal pool at the end of the program (eg instruction in the line number 55 started next).

```
17                              LTORG
18      002D        *           =C'EOF'                  454F46
:
54                              END         FIRST
55      1076        *           =X'05'                   05
```

10.List out the some assembler directives.

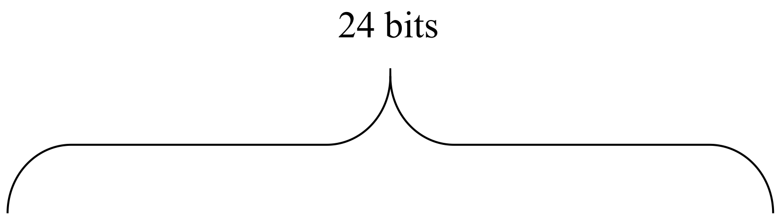
Name of the Assembler Directive	**Function**
START	Specifies name & starting address of the program
END	Indicates End of the source program
BYTE	Generates character or hexadecimal constant,occupying as many bytes as needed to represent the constant.
WORD	Generates one-word integer constant
RESB	Reserves the indicated number of bytes for a data area.
RESW	Reserves the indicated number of words for a data area.

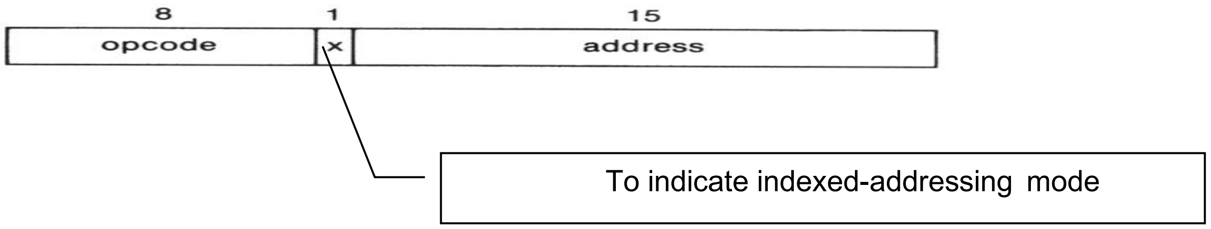
Part – B (11 Marks)

11.a. Draw the format of an instruction and explain.(5)

SIC - Instruction formats :

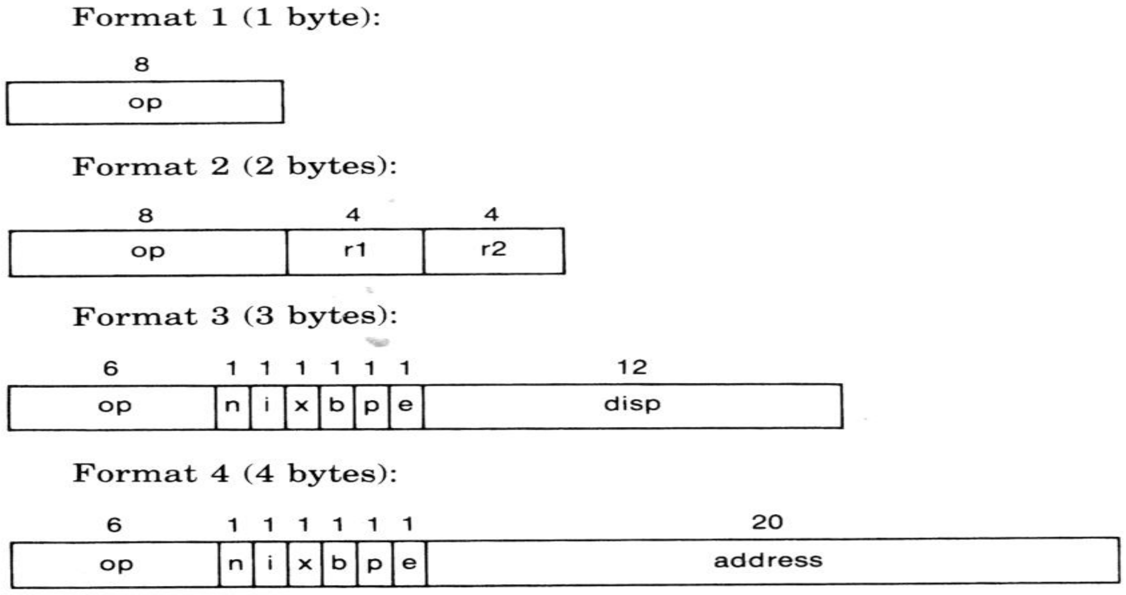
All machine instructions on the standard SIC have the following 24-bit format





SIC/XE - Instruction formats

Maximum memory in SIC/XE system is 1 megabyte(2^{20} bytes),address will no longer fit into a 15-bit field so, the instruction format used on the standard version of SIC is not suitable (SIC Instruction format uses 15-bit address, SIC/XE require 20-bit address)



Instruction that reference memory uses. In addition, SIC/XE provide two more instruction format(format 1, format 2) that do not reference memory at all.

CISC- Complex instruction set computers

Vax- **Instruction formats**

VAX m/c instructions use variable length instruction format.

Each instruction consists of operation code followed by up to 6 operand specifiers which depend on the type of instruction. Each specifier give information to locate the operand.

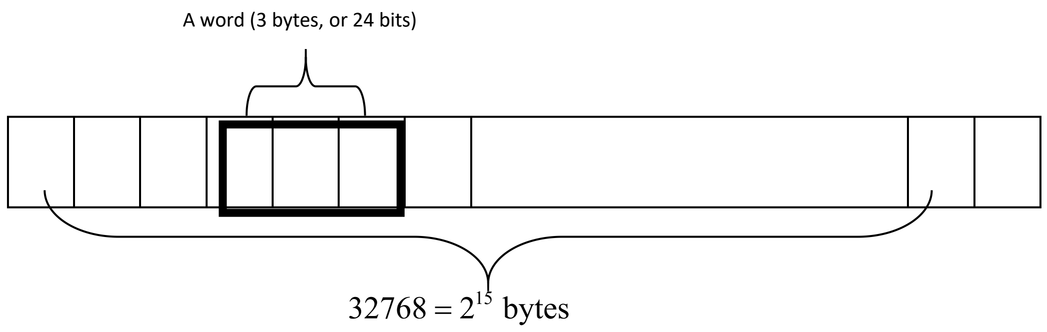
b. Describe the machine structure. (6)

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
    - Instruction formats
    - Addressing mode
    - Registers
- **Machine independency of system software**
  - General design and logic is basically the same:
    - Code optimization
    - Subprogram linking
- Two versions
  - Standard model (SIC) and an XE version (SIC/XE)
  - Upward compatible
    - Programs for SIC can run on SIC/XE

SIC Architecture

- **Memory**
  - It consist 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.

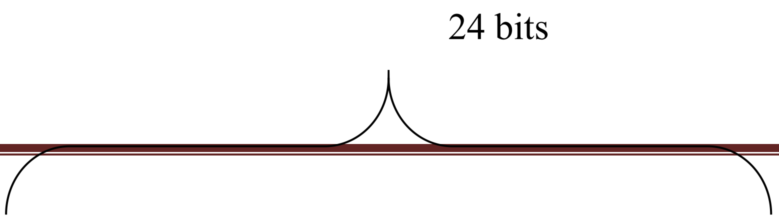


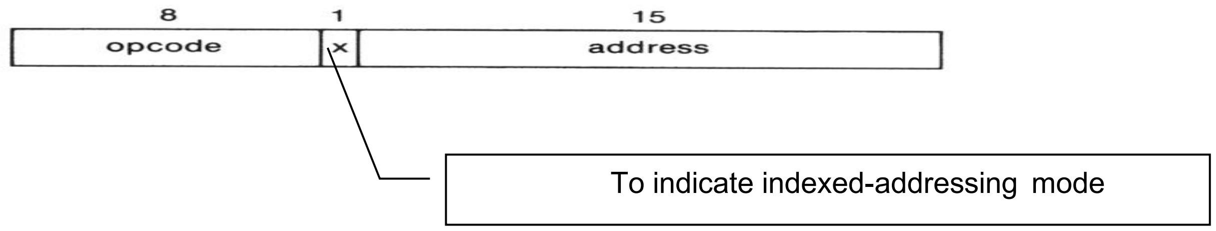
- Registers
- Five 24-bit registers

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; 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**
  Characters: 8-bit ASCII codes
  Integers: 24-bit binary numbers

  2's complement for negative values
  Floating-point numbers: No
- **Instruction formats**:
  All machine instructions on the standard SIC have the following 24-bit format





Addressing modes:
Two addressing modes:

1. Direct
2. Indexed

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

( ): Contents of a register or a memory location

Target address calculation for direct addressing mode is

TA= 2001

Target address calculation for indexed addressing mode is

TA= 2000 + (X)

= 2000+1

= 2001

- **Instruction set**

SIC provides a basic set of instructions that are sufficient for most simple tasks

- Load and store registers
    - LDA, LDX, STA, STX
- Integer arithmetic operations (involve register A and a word in memory, save result in A)
    - ADD, SUB, MUL, DIV
- Comparison instruction
- Comparison instruction

COMP instruction compares the value in the register A with another value of a variable and set the condition code CC to indicate the accumulator value is (<, =, >) the other values of a variable or word in the memory with which it is compared.

- Conditional jump instructions (according to CC)
    - JLT (Jump Less Than), JEQ(Jump Equal To), JGT(Jump Greater Than) instruction test the setting of CC and jumps accordingly.

- Eg.

```
            :
            :
            COMP   B { compare value of B with the accumulkator
            JLT       WLOOP {if accumulator content is < B jump to WLOOP}
WLOOP                 ADD     INCR { Add the value of INCR to the accumulator}
```

- Subroutine linkage
    - JSUB (jumps and places the return address in register L)
    - RSUB (returns to the address in L)

- **Input and output**

Read data (RD) or Write data (WD) instructions are used to perform input or output operations.

Program, which has to perform data transfer, has to wait until the device is ready and then execute the RD or WD.

Input and output operations are performed by transferring 1 byte to or from the rightmost byte of register A

Eg.

```
                    RD      INPUT {Read character into register A}
                    :
                    :
```

- **Test Device Instruction**

Test Device Instruction (TD) test whether the device is ready to send or receive data. The CC contains the result of this test. (If the condition code (CC) is set for '<' it implies that the device is ready and CC set for '=' indicates that the device is not ready)

```
RLOOP          TD      INPUT  {test the input device to see if it is ready}

          JEQ     RLOOP{if the input device is not ready jump to' RLOOP'}

          :

          :

INPUT          BYTE   X'F1'    {X'F1'   {F1 is the code for input device}
```

- Some features of the assembler that are not closely related to m/c architecture are called m/c independent assembler features.
- Such fetures are generally found on most large and complex machines.

Literals:

- It is convenient for a programmer to write the value of a constant operand as a part of the instruction that uses it.
- Such an operand is called literal because its value is stated "literally" in the instruction.
- Literal is identified by the prefix '='.

```
11        001A      ENDFIL      LDA       =C 'EOF'        032010
:
:
:
47        1062      WLOOP       TD        =X'05'          E32011
```

- The Literal in the line number 11 given above specifies a three byte operand whose character string is EOF.
- |||ly the line number 46 specifies a one byte literal with the hexadecimal value 05.

Advantage:

1. Literal avoids having to define constant else where in the program and make up a label for it.

Difference between literal and an immediate operand

Immediate operand	Literal
The operand value is assembled as a part of machine instruction.	The assembler generates the specific value of the operand as a constant and stores it at some other memory location.The address of this generated constant is used as the target address for the machine instruction.

Literal Pools:

- All literal operands used in a program are gathered into one or more literal pools
- Literals are placed into the literal pool at the end of the program (eg instruction in the line number 55 started next).

```
17                                    LTORG
18          002D        *             =C 'EOF'                    454F46
:
54                                    END           FIRST
55          1076        *             =X '05'                      05
```

Literal Pools Listing Examples:

LTORG Assembler Directive

- Sometimes literals are placed into a pool at some other location in the object program,other than the end of the program.
- To allow this we introduce the assembler directive LTORG.
- When LTORG is used the assembler creates a literal pool that contains all the literal operands used since the previous LTORG or beginning of the program (eg line no 18).
- Literals placed using LTORG will not be repeated at the end of the program.

Advantage of LTORG directive

- Literal pool is placed too far away from the instruction referring it when LTROG assembler directive is not used. Thus the need for LTORG arises when it is desirable to keep the literal operand close to the instruction that uses it.

Duplicate Literal

Most assembler recognizes literals.

For eg:

```
47        WLOOP      1062      TD          =X ‘05’      E32011
:
:
50                        106B        WD             =X ‘05’
DF2008
```

In the above case the literal =X ‘05’ is used on lines 47 and 50.However only one data area with this value is generated.thus both instructions refer the same address in the literal pool for their operand.

How does the assembler Handles Literal Operand?

- The basic data structure needed for the assembler to handle literal operand is LITTAB.

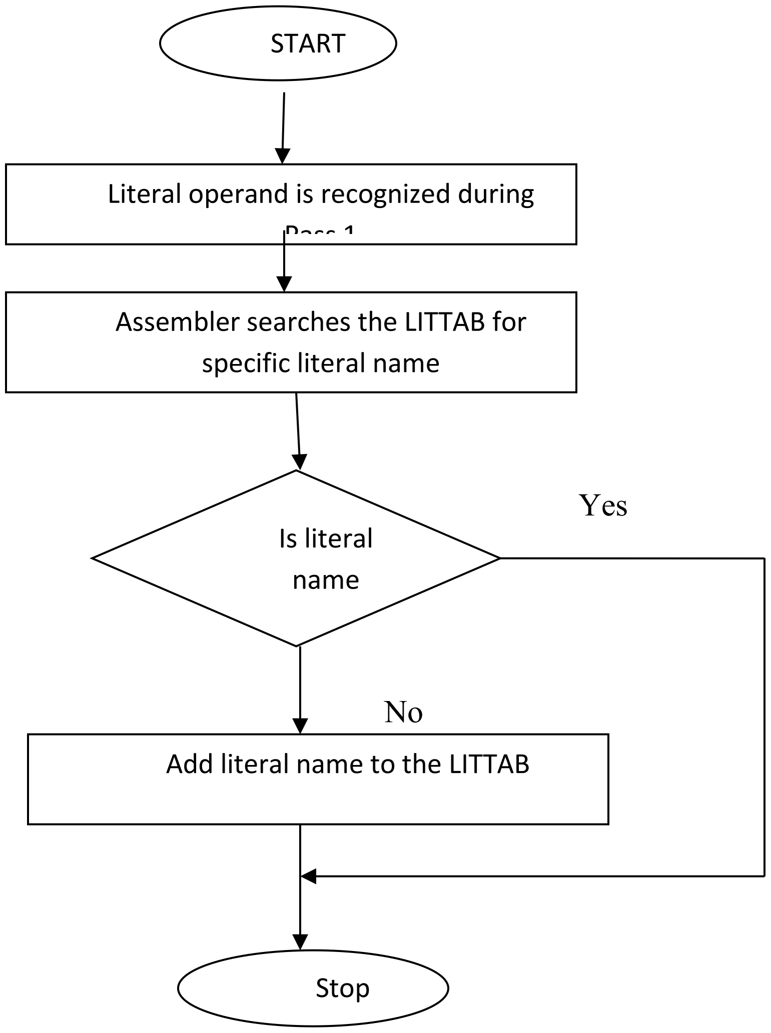
LITTAB Contains

1. Address assigned to the operand when it is placed in a literal pool.
2. Literal name
3. Operand value
4. Length

PASS1

- As each literal operand is recognized during pass1 the assembler reaches the LITTAB specifies literal name.
- If the literal is already present no action,otherwise it is added to the LITTAB.

Adding literal name to a LITTAB


START
Literal operand is recognized during Pass 1
Assembler searches the LITTAB for specific literal name
Is literal name
Yes
No
Add literal name to the LITTAB
Stop

- When pass1 encounter a LTORG,statement or end of program the assembler makes a scan of the literal table.
- At this time literal currently in the table,is assigned an address,unless such an address has already been filled in.
- As each address is assigned, the location counter is update to reflect the number f byte occupied by each literal.

PASS 2

- Operand address is got from the LITTAB for each literal operand encountered.
- The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD statements.
- If literal value represents an address in the program ,the assembler must also generated the appropriate modification record.

Symbol Defining Statements:

- Most assemblers provide an assembler directive that allows programmer to define symbols and specify their values.
- The assembler directive generally used is EQU.

Syntax:

```
Symbol      EQU       Value
```

Use of EQU

It establishes symbolic names that can be used for improving the readability of the statement.

For eg consider the following statement

```
+LDT                    #4096
```

- The statement loads the value 4096 into an integer.However the meaning of it is not clear.
- To make it more informative the assembler directive EQU can be used as specified next.

```
17        MAXLEN          EQU         4096
:
:
                           +LDT       #MAXLEN
```

- When the assembler encounters EQU,”MAXLEN” and “4096” are entered into the SYMTAB.
- During assembly of the LDT instruction,the assembler searches SYMTAB for the symbol MAXLEN,and uses its value as the operand in the instruction.

Advantages:

- The resulting object code is exactly the same as the original version of the instruction however the source statement is easier to understand.

- If we want to change the maximum length it is easier to change MAXLEN value rather than searching the whole program for all the statements where #4096 is used.

Assembler directive EQU

- In instructions like the assembler expects register numbers 0 and 1 instead of the registers A and X respectively.

```
RMO     0,1
```

- In such cases the assembler directive EQU can be used so that A and X can be recognized as '0' and '1' by the assembler.

- That is the programmer could include the sequence of statement like

```
A          EQU        0
X          EQU        1
L          EQU        2
```

In this case A,X,L are entered into the SYMTAB with values 0,1,and 2 respectively.the assembler will search the values for A,X,L from the SYMTAB and then assemble the instruction,of the form

```
RMO          A,X
```

- If the register R0,R1 etc are present,some of them can be used as BASE registers,some as index registers.
  Statements which follows can be used,for used ,for such purpose

```
BASE      EQU      R1
INDEX     EQU      R2
```

Assembler directive ORG

- The directive called ORG (for "origin") is of the form **ORG value**,in which value may be constant or an expression with constants or symbols.
- When this statement is encountered the assembler resets its location counter to the specified value.
- The ORG statement will affect the location of all the statement which follows them until the next ORG statement is encountered.

- Suppose we hve symbol table (STAB) with the following structure.
  STAB(100 entries)

Symbol 6 Bytes	value 3 bytes	Flags 2 bytes

- The SYMBOL fields contains a 6 byte user-defined symbol.
- “VALUE” is a one word (3 byte) representation of the value assigned to the symbol.
- “FLAGS” is a 2 byte field that specifies symbol type and other information.
  We can reserve space for the table using the following statement

```
STAB              RESB          1100
```

- Entries in the table can be referred using indexed addressing.
- Index register holds the offset of the desired entry.
- If we want to refer to the fields SYMBOL,VALUE and FLAGS individually, we must also define labels using the following EQU statements.

```
SYMBOL          EQU               STAB
                              VALUE
  EQU             STAB+6
  FLAGS           EQU               STAB+9
```

This would allow us to write for eg

```
LDA        VALUE ,X
```

To fetch the VALUE field from the table entry indicated by the content of register X.

- We can accomplish the same symbol definition using ORG in the following way:

```
STAB        RESB       1100        → Reserve '1100' bytes for the symbol table.
            ORG        STAB        →Resets the location counter to the beginning
                                     address of the table
SYMBOL      RESB        6          →Starting location of SYMBOL is STAB (beginning address
                                     of the table)and it occupies 6 bytes.
VALUE       RESB        3          →Starting location of VALUE is STAB+6 and it occupies 3
                                      Bytes
FLAGS       RESB        2          → Starting location of FLAGS is STAB+9 and it occupies 2
                                      Bytes
            ORG        STAB+1100 →This sets the LOCCTR back to the address of next
                                      unsigned byte of memory after STAB.
```

- In some assemblers previous value of STAB is already known so just the ORG statement is sufficient instead of
-

```
ORG           STAB+1100
```

Restrictions in using EQU and ORG statements

In case of EQU

- Any symbol can be used only after defining it.
  (e.g) the sequence

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

Would be allowed where else the sequence

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

Is not allowed.

In case of ORG

- All symbols used to specify the new location counter value must have been previously defined.
  e.g:The sequence

```
ORG         STAB
STAB         RESB         1100
```

Would not be permitted.

Expressions:

- So far assembler language statements have used labels,literals etc as instruction operands.most assemblersallow the use of expressions which when evaluated,gives the operand address or value.
- Assembler allow arithmetic expressions with the operators "+","-","*" and "/".
- Expressions may have user defined symbols,constants or special symbols.
  For eg consider the following statement

```
BUFFER            RESB          4096
BUFFEND           EQU            *
```

- It gives BUFFEND a value that is the address of next byte after the BUFFER area.
- Some values in the object program are relative to the beginning address (eg labels)while some others absolute (insependent of the program location eg constants).
- Values of expressions are either relative or absolute.
- The symbol whose value is given by EQU may be an absolute or relative term,depending upon the expression used to define its value.

Expressions are classified depending on the value they produce as

1. Absolute Expression
2. Relative expression

Comparison of absolute & Relative Expression

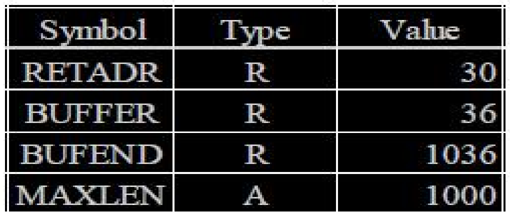
Absolute Expression	Relative expression
An expression with only absolute terms is called expression. it may also contain	A relative is one in which all of the relative terms except one can be paired,

relative terms provided the relative terms occur in pairs and the terms in such pair have opposite signs.	the remaining unpaired relative term must have a positive sign expression .
None of the relative terms may enter into a multiplication or division operation	None of the relative terms may enter into a multiplication or division operation

Eg:

```
MAXLEN          EQU          BUFFEND-BUFFER
```

- In this example BUFFEND and BUFFER represent an address within the program.
- Difference b/w two addresses is the length of the buffer area,which is an absolute term.
- To determine the type of expression we must keep track of the types of all symbols.
- For this purpose we need a flag in the symbol to indicate the type of value,in addition to the value itself.
- Thus have symbol table may contain the following details:



Symbol	Type	Value
RETADR	R	30
BUFFER	R	36
BUFEND	R	1036
MAXLEN	A	1000

- Using the type field ,the assembler can easily determine the type of each expression used as an operand and generate the modification record for relative values.

Program blocks:

- In the example seen so far the program being assembled was treated as a unit.
- The source program logically contained subroutines,data ares etc.however the assembler handled them as one entity resulting in a single block of object code.

Definition:

Program blocks refers to segments that are rearranged within a single object program unit.

Control Sections:

Control section refers to segments that are translated into independent object program unit.

Program block:

The 3 blocks used to write programs using program blocks are:

1. **Unnamed default block**
   Contains executable instruction of the program
2. **CDATA block**
   Contains all the data areas that are a few words or less in length.
3. CBLKS
   Contains all the data area that consists of larger blocks of memory.

BLOCK NAME	BLOCK NO
DEFAULT	0
CDATA	1
CBLKS	2

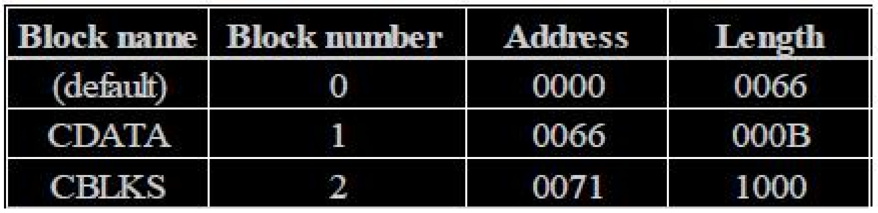
Program:

- The assembler directive 'USE"is used to indicate which portion of the source belongs to which block.
- If no USE statements are used the program belongs to the unnamed default block
- Line number
  - ✓ 1 indicate the beginning of the default block
  - ✓ 15 indicate the beginning of CDATA block
  - ✓ 18 begin CBLKS block.
  - ✓ 25 resumes default block
  - ✓ 40 resumes CDATA block.
  - ✓ 45 resumes default block
  - ✓ 40 resumes CDATA block.

PASS 1

- Assembler will logically rearrange the segments of program block to gather together the pieces of each block.
- These blocks will then be assigned address in the object program.
- There is a separate location counter for each block.
- At the beginning of the program the LC=0.

- When switching to another block current value of LC is saved and it is restored when the block is resumed.
- When labels are entered into the symbol table the block name or number is stored along with the assigned relative address.
- At the end of pass1 the LC of each block will tell the length of the block.
- The end of pass1 creates the following table:



Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

Note:

Starting address of ‘CDATA’ is 0066,and its length is ‘000B’

- Decimal equivalent of 0066 is

0066

$\longrightarrow 6 * 16^0 = 6$

$\longrightarrow 6 * 16^1 = \underline{96} +$

$\underline{102}$

- Decimal equivalent of 000B is

000B

$\longrightarrow 11 * 16^0 = 11$

$\longrightarrow 0 * 16^1 = \underline{0} +$

$\underline{11}$

- 102+11=113
- Hexadecimal equivalent of 113 is

16 | 113 =71(starting location of next block)

7 $\longrightarrow$ 1

- Thus the starting address of “CBLKS”block is ‘0071’ and its lengt is ‘1000’

- Starting address of next block,if its exists is 0071+1000=1071.but as no other block follows "CBLKS" the ending address of "CBLK" is "1070".

PASS 2

The assembler needs the address of each symbol relative to the start of the object program.so it adds the location of the symbol with the start address of the individual block.

(e.g) if the CDATA starting address is 0066,and the following statement is an instruction within the block.

0003	LENGTH

The desired target address of LENGTH is

0066+0003=0069

- The loader will simply load the object program at the indicated addresses.
- The following table represents the different blocks used in the program along with the details of the address locations occupied by them:

Block Name	**Locations occupied**
Default	0000→0065
CDATA	0066→0070
CBLKS	0071→1070

Program blocks traced through the assembly and loading process

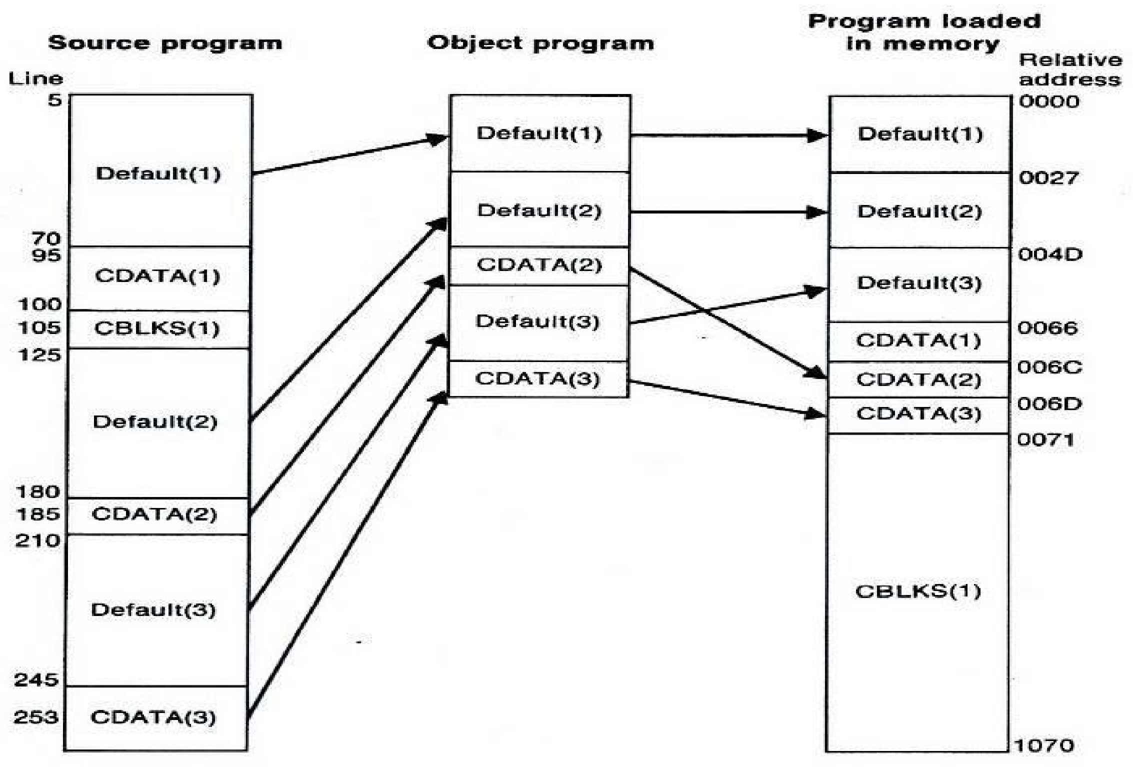


Figure 2.14 Program blocks from Fig. 2.11 traced through the assembly and loading processes.

Advantage of separating the program into blocks:

- Separation of programs into blocks considerably reduces the addressing problem.as the large buffer area is moved to the end of the object program,we no longer need to use extended format instructions.
- The base register is no longer necessary,so LDB and BASE statements are deleted from the source code.
- The problem of placement of literals in the program is easily solved.we simply include the LTORG statement in the CDATA block to be sure that the literals are placed ahead of any large data areas.
- The machine may suggest the object program to appear in the memory in a particular order,but the programmer may wish the source program to appear in a different order,both the man and machine wishes are satisfies by the use of program blocks,with the assembler providing the required reorganization.

Control Sections and Program Linking

A control section is a part of the program that maintains its identity after assembly.

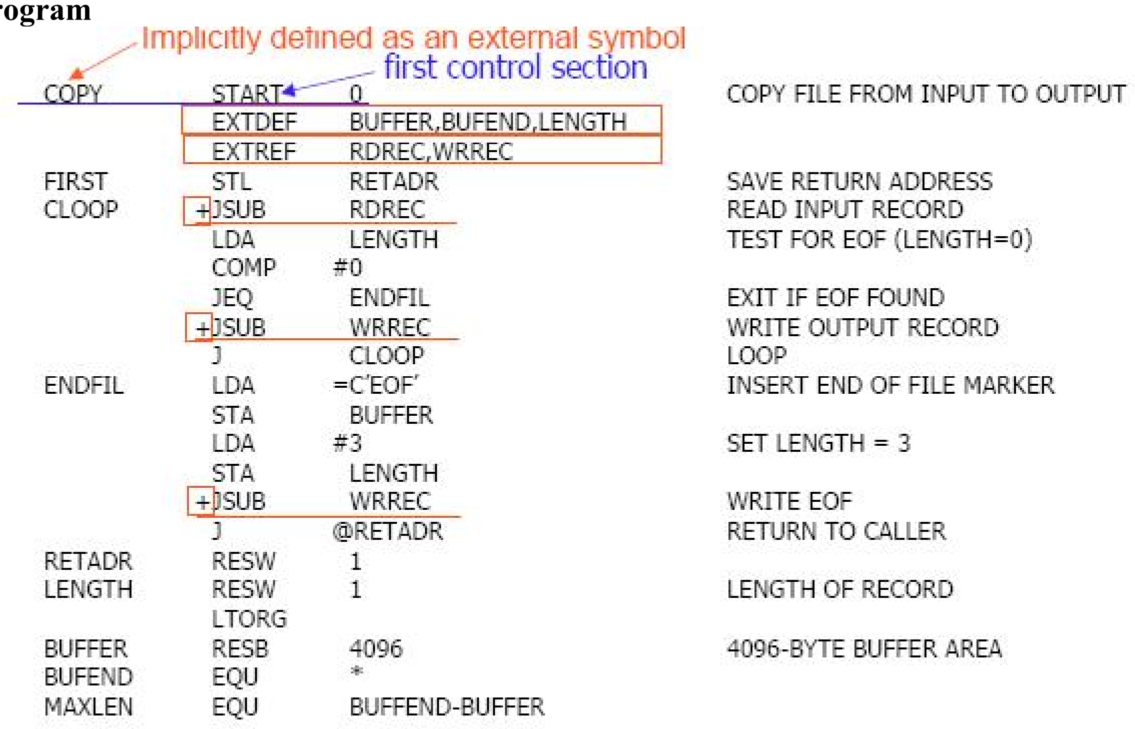
- the programmer can assemble, load, and manipulate each of these control sections separately

- Different control sections are most often used for subroutines or other logical subdivisions of a program
- Flexibility is a major benefit of using control sections as the programmer can assemble, load, and manipulate each of these control sections separately
- The control sections are independently loaded so the assembler does not know where the other section will be loaded at execution time.
- instruction in one control section may refer to another section instructions or data such references between control sections are called external references.
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

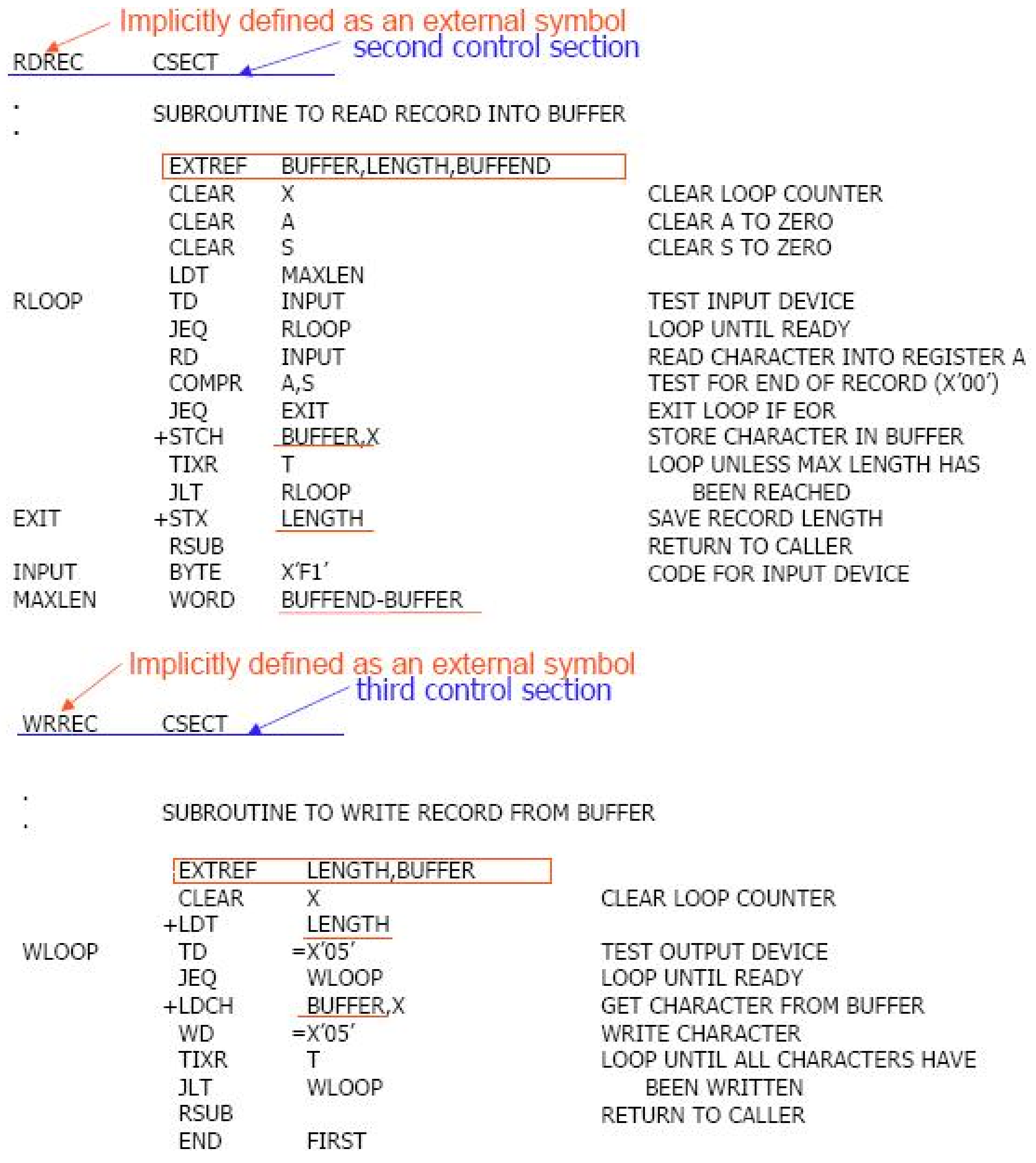
In the following program there are 3 control sections:

- Main program
- Subroutine for reading
- Subroutine for writing

Program



```
COPY      START   0                      COPY FILE FROM INPUT TO OUTPUT
          EXTDEF  BUFFER,BUFEND,LENGTH
          EXTREF  RDREC,WRREC
FIRST     STL     RETADR                 SAVE RETURN ADDRESS
CLOOP     +JSUB   RDREC                  READ INPUT RECORD
          LDA     LENGTH                 TEST FOR EOF (LENGTH=0)
          COMP   #0
          JEQ     ENDFIL                 EXIT IF EOF FOUND
          +JSUB   WRREC                  WRITE OUTPUT RECORD
          J       CLOOP                  LOOP
ENDFIL    LDA    =C'EOF'                 INSERT END OF FILE MARKER
          STA     BUFFER
          LDA    #3                      SET LENGTH = 3
          STA     LENGTH
          +JSUB   WRREC                  WRITE EOF
          J      @RETADR                 RETURN TO CALLER
RETADR    RESW    1
LENGTH    RESW    1                      LENGTH OF RECORD
          LTORG
BUFFER    RESB    4096                   4096-BYTE BUFFER AREA
BUFEND    EQU     *
MAXLEN    EQU     BUFFEND-BUFFER
```


Implicitly defined as an external symbol
second control section
Implicitly defined as an external symbol
third control section

```
RDREC       CSECT
.
.           SUBROUTINE TO READ RECORD INTO BUFFER
.
            EXTREF   BUFFER,LENGTH,BUFFEND
            CLEAR    X                             CLEAR LOOP COUNTER
            CLEAR    A                             CLEAR A TO ZERO
            CLEAR    S                             CLEAR S TO ZERO
            LDT      MAXLEN
RLOOP       TD       INPUT                         TEST INPUT DEVICE
            JEQ      RLOOP                         LOOP UNTIL READY
            RD       INPUT                         READ CHARACTER INTO REGISTER A
            COMPR    A,S                           TEST FOR END OF RECORD (X'00')
            JEQ      EXIT                          EXIT LOOP IF EOR
           +STCH     BUFFER,X                      STORE CHARACTER IN BUFFER
            TIXR     T                             LOOP UNLESS MAX LENGTH HAS
            JLT      RLOOP                             BEEN REACHED
EXIT       +STX      LENGTH                        SAVE RECORD LENGTH
            RSUB                                   RETURN TO CALLER
INPUT       BYTE     X'F1'                         CODE FOR INPUT DEVICE
MAXLEN      WORD     BUFFEND-BUFFER

WRREC       CSECT
.
.           SUBROUTINE TO WRITE RECORD FROM BUFFER
.
            EXTREF   LENGTH,BUFFER
            CLEAR    X                             CLEAR LOOP COUNTER
           +LDT      LENGTH
WLOOP       TD      =X'05'                         TEST OUTPUT DEVICE
            JEQ      WLOOP                         LOOP UNTIL READY
           +LDCH     BUFFER,X                      GET CHARACTER FROM BUFFER
            WD      =X'05'                         WRITE CHARACTER
            TIXR     T                             LOOP UNTIL ALL CHARACTERS HAVE
            JLT      WLOOP                             BEEN WRITTEN
            RSUB                                   RETURN TO CALLER
            END      FIRST
```

Control sections

- can be loaded and relocated independently of the other 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
- Because of this, there should be some means for linking control sections together assembler directive: CSECT
  secname CSECT
- separate location counters for each control section

External Definition and Reference

- Instructions in one control section may need to refer to instructions or data located in another section

External definition

EXTDEF name [, name]

EXTDEF names symbols that are defined in this control section and may be used by other sections

Ex: EXTDEF BUFFER, BUFEND, LENGTH

External reference

EXTREF name [,name]

EXTREF names symbols that are used in this control section and are defined elsewhere

Ex: EXTREF RDREC, WRREC

To reference an external symbol, extended format instruction is needed.

External Reference Handling

Case 1

- 5 0003 CLOOP +JSUB RDREC 4B100000
- The operand RDREC is an external reference.
- The assembler
  - ✓ Has no idea where RDREC is
  - ✓ Inserts an address of zero
  - ✓ Can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

Case 2

- 41 0028 MAXLEN WORD BUFEND-BUFFER 000000
- There are two external references in the expression, BUFEND and BUFFER.
- The assembler
  - inserts a value of zero
  - passes information to the loader

- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

- Case 3

  BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.
- 22 1000 MAXLEN EQU BUFEND-BUFFER

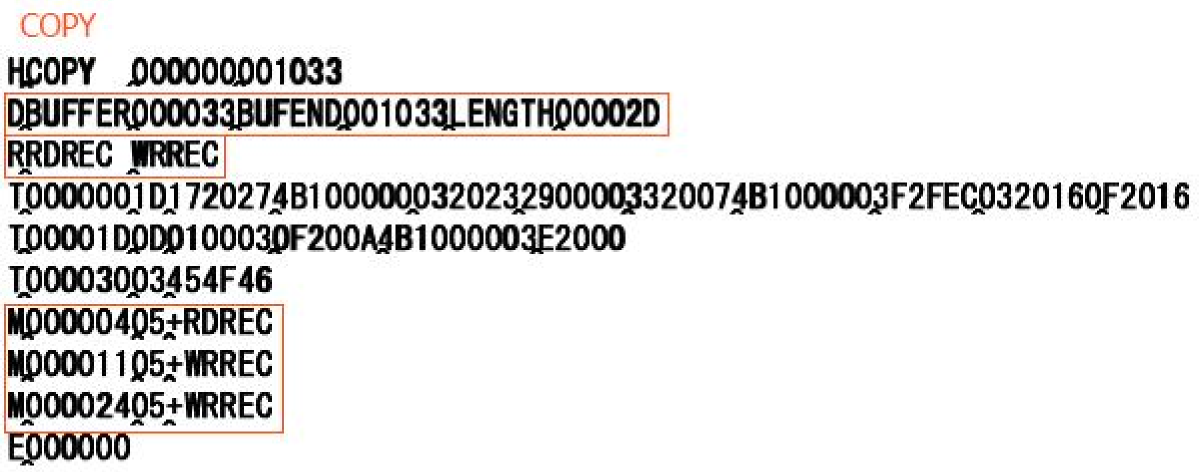
Record for object program

- *Define record*
  - Col. 1 D
  - Col. 2-7 Name of external symbol defined in this control section

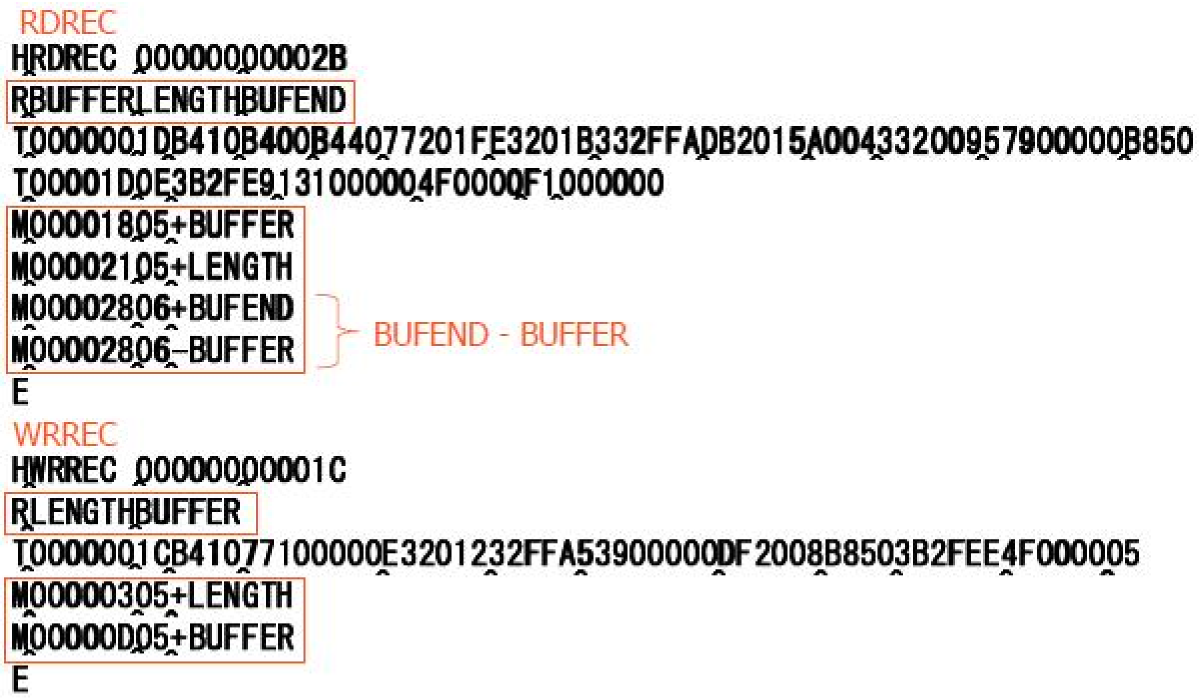
- Col. 8-13 Relative address within this control section (hexadeccimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

- *Refer record*
    - Col. 1 D
    - Col. 2-7 Name of external symbol referred to in this control section
    - Col. 8-73 Name of other external reference symbols
- Modification Record
    - *Modification record*
    - Col. 1 M
    - Col. 2-7 Starting address of the field to be modified (hexiadecimal)
    - Col. 8-9 Length of the field to be modified, in half-bytes (hexadeccimal)
    - Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

Object Program



```
COPY
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000
```



```
RDREC
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER      BUFEND - BUFFER
E
WRREC
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

May 2012

Part – B (11 Marks)

1. List out and discuss the machine dependent assembler features.

MACHINE DEPENDENT FEATURES OF ASSEMBLER

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

- Instruction formats
- Addressing modes
- Program relocation

INSTRUCTION FORMAT AND ADDRESSING MODES:

FORMAT 1:

Length = 1 byte

8bits

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

Displacement = address of operand – [PC]   -2048<=displacement<=2047

BASE – relative displacement Calculation

Displacement = address of operand – [base] displacement<=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

If the instruction contains only opcode, assembler takes Format-1. If the operands are registers then it takes Format-2. Otherwise it takes Format-3 for translating the instruction.

In Format-3, assembler first attempt to translate the instruction using program counter relative addressing

If the displacement calculated using PC relative is less than 2047 and greater than -2048, base relative addressing is used to translate the instruction

If the displacement calculated using base-relative is greater than 4095 then, it is extended format (if + is appeared as prefix of the instruction) is used to translate the instruction.

ADDRESSING MODES:

Direct addressing

@ Indirect addressing

# Immediate addressing

Direct and indirect addressing can be combined with indexing

- Format of instruction has specified in object code using flag e
  - e =1 → extended format (format 4)
  - e =0 → format 3
- Mode of displacement calculation has specified in object code through the flags p and b
  - p=1 → PC relative mode
  - b=1 → base relative mode
- Addressing mode has specified in object mode code through the flags n and i.
  - n=0&i=0 → direct addressing (or) n=1& i=1.
  - n=0&i=1 → immediate addressing
  - n=1&i=0 → indirect addressing

PROGRAM RELOCATION:

It is desirable to have more than one program at a time sharing the memory and resources of the machine. If we knew in advance exactly which programs were to be executed concurrently, we could assign addresses when the programs were assembled. So that they would fit together without overlap or wasted space. But the assembler does not know the actual location where the program will be loaded and it cannot make changes in the address used by the program.

It is desirable to be able to load a program into memory wherever there is room for it. In such a situation the actual starting address of the program is not known until load time.

But 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.

The only instructions of the program that require modification at load time are those that specify direct addresses. So only this part of the program, the assembler inserts the modification record in the object program. So, the loader can identify the instruction which needs the modification. The loader can easily load the program into the memory where it finds the free space.

In object program, the text records are exactly same as those that would be produced by an absolute assembler. The text records are interpreted as relative locations. There is one Modification record for each address field that needs to be changed when the program is relocated.

The format of Modification Record is

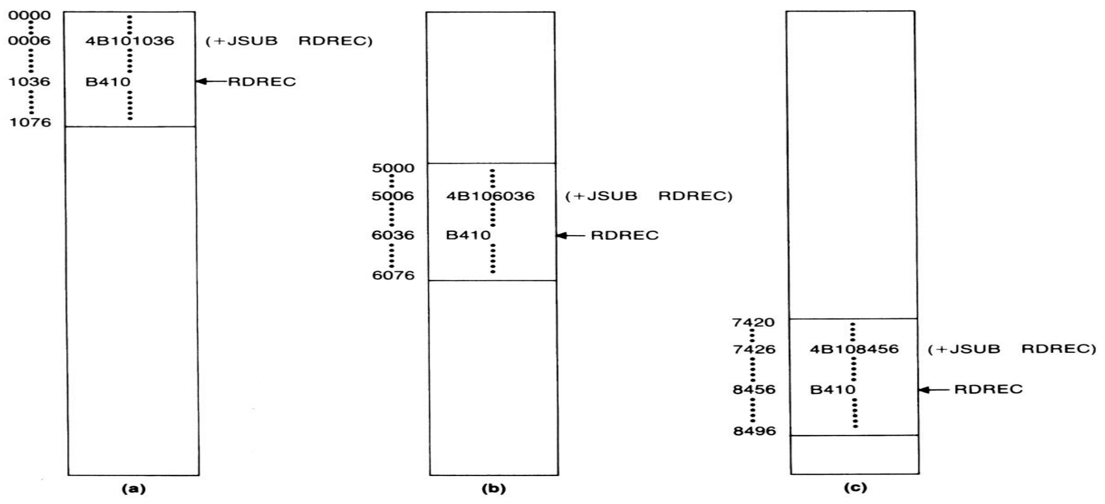
MODIFICATION RECORD:

Col 1	M
Col 2-7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
Col8-9	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.

Now suppose that we want to load this program beginning at address 5000 (Fig b). The address of the instruction labeled RDREC is then 6036. Thus the JSUB instruction must be modified as shown to contain this new address. Likewise, if we loaded the program beginning at address 7420 (Fig. c), the JSUB instruction would need to be changed to 4B108456 to correspond to the new address of RDREC.

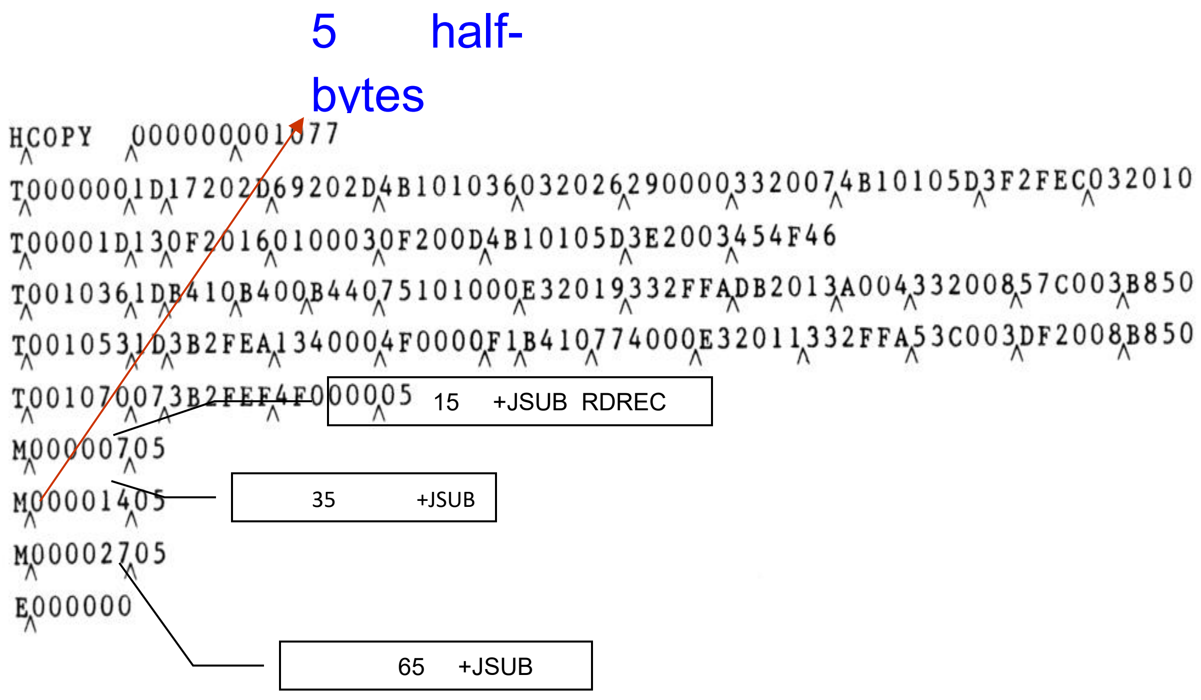
No matter where the program is loaded, RDREC always 1036 bytes past the starting address of the program. This means that we can solve the relocation problem in the following way:

1. When the assembler generated the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program.

2. The assembler will also produce a command for the loader, instructing it to add the beginning address of the program to the address field in the JSUB instruction at the load time.

The command for the loader is achieved through writing Modification Record in the Object Program.

RELOCATABLE OBJECT PROGRAM



```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

2. Briefly explain about assembler design options.

- One pass assembler
- Two pass assembler with overlay structure
- Multi pass assembler

Two-Pass Assembler with Overlay Structure:

- *For small memory*
    - pass 1 and pass 2 are never required at the same time
    - three segments
        - ✓ root: driver program and shared tables and subroutines
        - ✓ pass 1
        - ✓ pass 2
        - ✓ tree structure
    - overlay program

One- pass assembler is used when we want to avoid the 2nd pass.

MULTI PASS assembler is used to handle forward reference during symbol definition

One-pass assemblers

Problem associated with forward reference

Operands are sometime s symbols that have not yet been defined in source program, when it is first encountered. Thus the assembler does not know what address to insert in the translated instruction, this is the main problem of using of forward reference in a program

Rectification for the problem of forward reference

1. Simply define the symbols in the source program before they referenced.
2. Practically avoiding such forward jumps is not very easy. So the assembler should adapt some other method in order to solve such reference problem.

Two type of one pass assembler

Type1 – **Load and go assembler** (object code is not written out, so

loader is not necessary)

Type2 – (object program is written out in a file so loader is necessary)

Type1 – **Load and go assembler**

- No object program is written out and so no loader is needed.
- Useful in system that involves program development and testing.

- Assembler simply generates object code as its scan the program, directly in memory for immediate execution.
- In the instruction operand is a symbol that has not yet been defined the operand address is omitted when the instruction is assembled.
- This symbol is entered into symbol table, an undefined symbol withg ‘*’.
- When the definition of the symbol is encountered, the proper address is inserted in a list associated with that symbol.
- When nearly half of the program translation is over some of the forward reference problems that existed earlier would have been resolved, while others might have been added.
- Continue this process to the end of the program to fill all the forward references properly.
- At the end of the program, symbol table entries still marked with ‘*’ are undefined symbol and should indicate error.
- If no errors occur until program termination the assembler searches the SYMTAB for the value of symbols and begins the execution of the program.
- For a load and go assembler the actual address must be known at assembly time.
- However the assembly process would be the same and the location counter will be assigned to the starting address of the program.

Line	LOC	Source statement			Object code
		Label	Opcode	operand	
1.	1000	COPY	START	1000	
2.	1000	EOP	BYTE	C ‘EOF’	454F46
3.	1003	THREE	WORD	3	000006
4.	1003	RETADR	RESW	10	
:	:	:	:	:	:

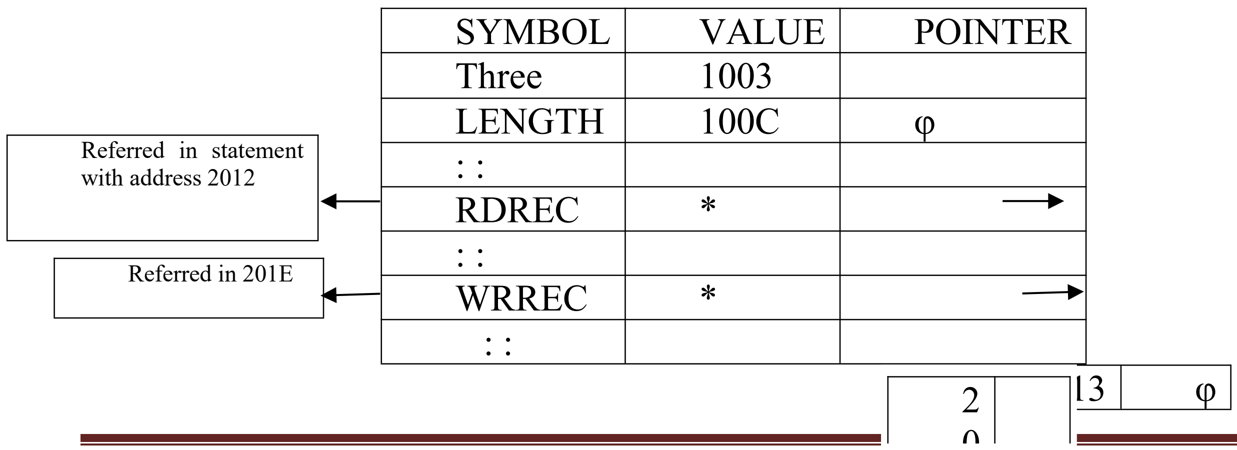
If the preceding statements in a program are processed by the load and go assembler the object code generated in the memory would be as follows:

Memory Address	contents			
1000	454f4600	0003xxxx	xxxxxxxx	xxxxxxxx
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:				
:				

Consider the following set of statements:

1	1003	THREE	WOR	3	000003
	:		D		
2	100	LENGT		1	
3	C	H	RESW	RDREC	48203
	2012	CLOOP	JSUB		D
4	:			WRRE	
	201E		JSUB	C	482062
5	:				
6	203	RDREC	LDX	ZERO	041006
	D	WRREC	LDX	ZERO	041006
	2062				

Symbol table used in a load and go assembler when the first four lines are processed is shown below:



SYMBOL	VALUE	POINTER
Three	1003	
LENGTH	100C	φ
: :		
RDREC	*	→
: :		
WRREC	*	→
: :		

Use:

- Load and go assembler is used when external working storage devices are not available to store intermediate file.
- It can also be used when the external storage is slow.

ADVANTAGES:

- It avoids the overhead of writing the object program out and then reading it back in.
- It is accomplished using one pass.
- Because the object program is produced in the memory rather than write out on secondary storage, handling of forward reference is less difficult.

Type 2

- Forward references are entered into the list as before however, en the definition of symbol is encountered instruction that made forward reference to that label will no longer be available in the memory for modification instead they would have been written out as part of a text in the memory for modification instead they would have been written out as part of a text record in the object program, with all zero in the place of any labels address location.
- Assembler must generate another text record with correct operand address
- Eg: T^0022013 ^02 ^ 203D is written to the object program when k=line number 5 is processed. Where 002013 is the starting address from which the object code (0000) following "48"
- Is to be modified "02" is the length of the text record (203d which succeds is 2 bytes long as each digit occupies ½ bytes)"203D" is the address to replace all zeros in the loaded object code "480000" generated earlier. Thus the object becomes 48203D

- Note: only modification record length ids represented with the number of half bytes. Text record length is always represented with the value in bytes.
- The loader will insert the actual address in to the instruction from the information available in this additional text record.
- When the program is loaded the correct address will replace all th e zeros previously loaded. Thus the final loaded object code is 48203D.

MULTI PASS ASSEMBLER:

IN ~~the~~ case of assembler directives like EQU that define symbols, any symbol used on right hand side must be defined previously in the program.

Eg. Two passes process the following sequence of code

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

During 1st pass.

1. ALPHA cannot be assigned as the value of BETA is not yet initialized.
2. BETA cannot be assigned a value, since DELTA is not defined before BETA.
3. Location of DELTA is entered in the symbol table.

During the next pass.

- Beta can be assigned with value of DELTA but ALPHA cannot be assigned, as the statement involving BETA comes after ALPHA.
- Multi pass assembler can make as many passes as are needed to process the definition are symbols.

- It is not necessary for an assembler to make more than two passes over th entire program instead the portion of the program that has forward reference is stored using pas
- s 1.
- Additional passes are made only through the stored definitions.
- This process is followed by a normal pass 2.

To solve forward reference problem

1. Store symbols that involve forward reference in the symbol table.

2. This symbol table also indicates which symbols are dependent on the value of others.

Eg: sequence of statements that involves forward reference.

```
1.  :       HALFSZ      EQU    MAXLEN/2
2.  :       MAXLEN      EQU    BUFFEND-BUFFER
    :
    :
    :
3.  1034    BUFFFER     RESB   4096
4.  2034    BUFFEND     EQU    *
```

The following figure show the symbol table entries resulting from pass1 processing of the first statement given below

variable	No. of undefined symbols in the expression	Value	Pointer to the list of symbols which depends on the variable
HALFSZ	&1	MAXLEN/2	Φ
MAXLEN	*		$\longrightarrow$

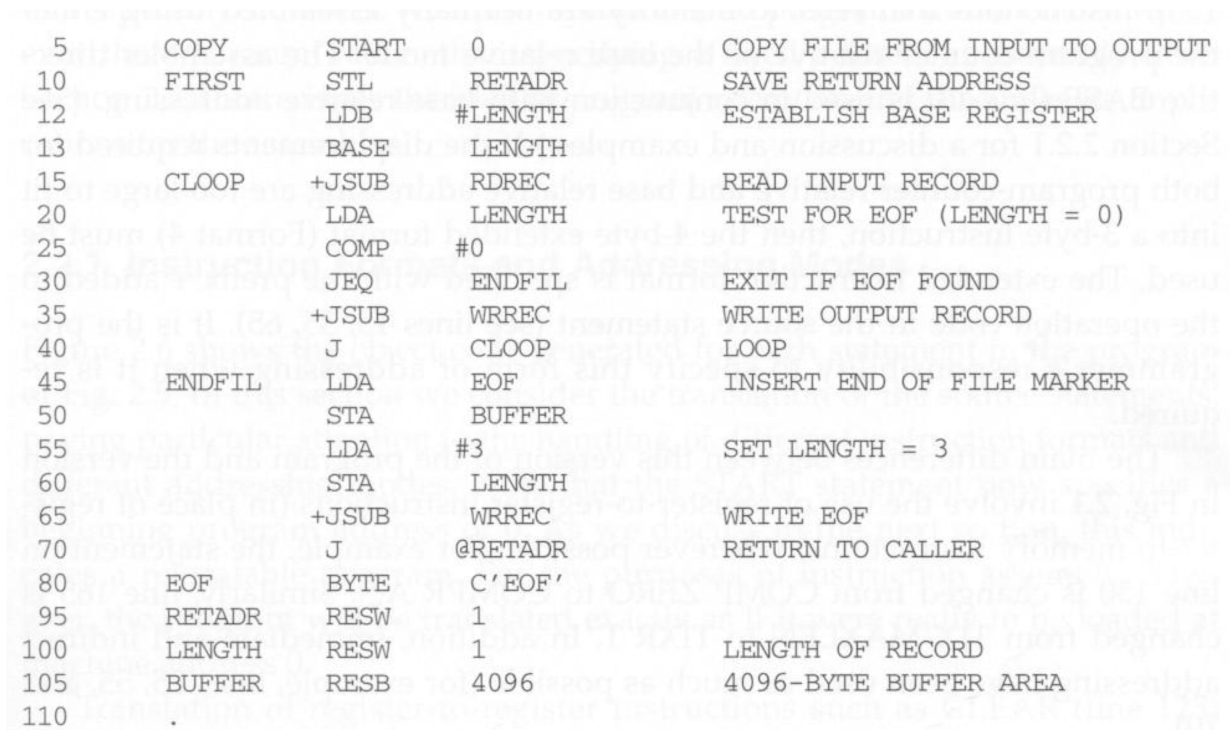
HALFSZ	φ

November 2012

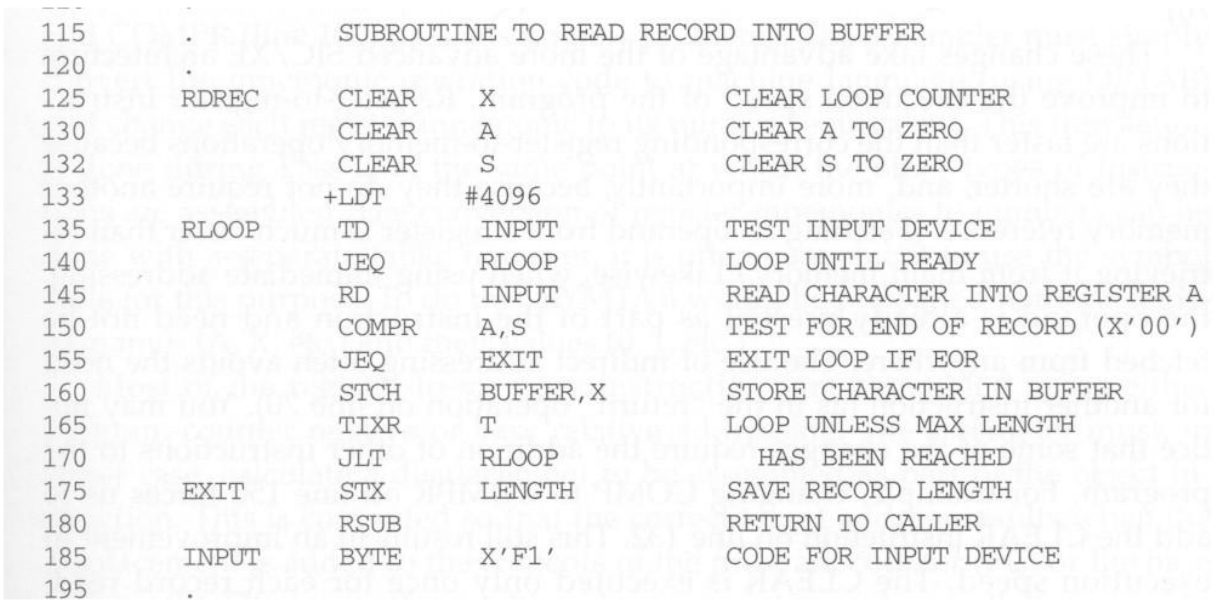
PART B 11mark

1. List out and discuss the machine dependent assembler features.

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



```
  5   COPY     START   0          COPY FILE FROM INPUT TO OUTPUT
 10   FIRST    STL     RETADR     SAVE RETURN ADDRESS
 12            LDB     #LENGTH    ESTABLISH BASE REGISTER
 13            BASE    LENGTH
 15   CLOOP   +JSUB    RDREC      READ INPUT RECORD
 20            LDA     LENGTH     TEST FOR EOF (LENGTH = 0)
 25            COMP    #0
 30            JEQ     ENDFIL     EXIT IF EOF FOUND
 35           +JSUB    WRREC      WRITE OUTPUT RECORD
 40            J       CLOOP      LOOP
 45   ENDFIL   LDA     EOF        INSERT END OF FILE MARKER
 50            STA     BUFFER
 55            LDA     #3         SET LENGTH = 3
 60            STA     LENGTH
 65           +JSUB    WRREC      WRITE EOF
 70            J       @RETADR    RETURN TO CALLER
 80   EOF      BYTE    C'EOF'
 95   RETADR   RESW    1
100   LENGTH   RESW    1          LENGTH OF RECORD
105   BUFFER   RESB    4096       4096-BYTE BUFFER AREA
110   .
```



```
        .       SUBROUTINE TO READ RECORD INTO BUFFER
        .
RDREC   CLEAR   X               CLEAR LOOP COUNTER
        CLEAR   A               CLEAR A TO ZERO
        CLEAR   S               CLEAR S TO ZERO
       +LDT     #4096
RLOOP   TD      INPUT           TEST INPUT DEVICE
        JEQ     RLOOP           LOOP UNTIL READY
        RD      INPUT           READ CHARACTER INTO REGISTER A
        COMPR   A,S             TEST FOR END OF RECORD (X'00')
        JEQ     EXIT            EXIT LOOP IF EOR
        STCH    BUFFER,X        STORE CHARACTER IN BUFFER
        TIXR    T               LOOP UNLESS MAX LENGTH
        JLT     RLOOP             HAS BEEN REACHED
EXIT    STX     LENGTH          SAVE RECORD LENGTH
        RSUB                    RETURN TO CALLER
INPUT   BYTE    X'F1'           CODE FOR INPUT DEVICE
        .
```

MACHINE DEPENDENT FEATURES OF ASSEMBLER

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

- Instruction formats
- Addressing modes
- Program relocation

INSTRUCTION FORMAT AND ADDRESSING MODES:

FORMAT 1:

Length = 1 byte

8bits

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

Displacement = address of operand – [PC] -2048<=displacement<=2047

BASE – relative displacement Calculation

Displacement = address of operand – [base] displacement<=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

If the instruction contains only opcode, assembler takes Format-1. If the operands are registers then it takes Format-2. Otherwise it takes Format-3 for translating the instruction.

In Format-3, assembler first attempt to translate the instruction using program counter relative addressing

If the displacement calculated using PC relative is less than 2047 and greater than -2048, base relative addressing is used to translate the instruction

If the displacement calculated using base-relative is greater than 4095 then, it is extended format (if + is appeared as prefix of the instruction) is used to translate the instruction.

ADDRESSING MODES:

Direct addressing

@ Indirect addressing

# Immediate addressing

Direct and indirect addressing can be combined with indexing

- Format of instruction has specified in object code using flag e
  - e =1 → extended format (format 4)
  - e =0 → format 3
- Mode of displacement calculation has specified in object code through the flags p and b
  - p=1 → PC relative mode
  - b=1 → base relative mode
- Addressing mode has specified in object mode code through the flags n and i.
  - n=0&i=0 → direct addressing (or) n=1& i=1.
  - n=0&i=1 → immediate addressing
  - n=1&i=0 → indirect addressing

PROGRAM RELOCATION:

It is desirable to have more than one program at a time sharing the memory and resources of the machine. If we knew in advance exactly which programs were to be executed concurrently, we could assign addresses when the programs were assembled. So that they would fit together without overlap or wasted space. But the assembler does not know the actual location where the program will be loaded and it cannot make changes in the address used by the program.

It is desirable to be able to load a program into memory wherever there is room for it. In such a situation the actual starting address of the program is not known until load time.

But 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.

The only instructions of the program that require modification at load time are those that specify direct addresses. So only this part of the program, the assembler inserts the modification record in the object program. So, the loader can identify the instruction which needs the modification. The loader can easily load the program into the memory where it finds the free space.

In object program, the text records are exactly same as those that would be produced by an absolute assembler. The text records are interpreted as relative locations. There is one Modification record for each address field that needs to be changed when the program is relocated.

The format of Modification Record is

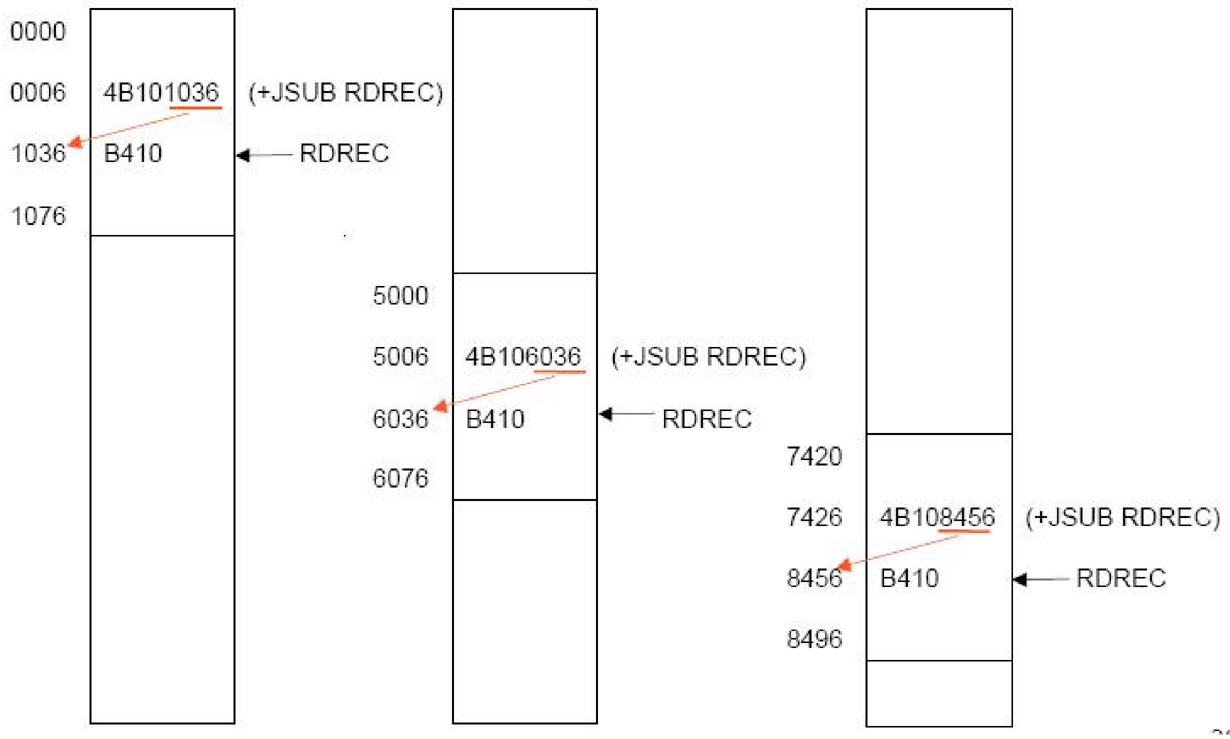
MODIFICATION RECORD:

Col 1	M
Col 2-7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
Col8-9	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.

Now suppose that we want to load this program beginning at address 5000 (Fig b). The address of the instruction labeled RDREC is then 6036. Thus the JSUB instruction must be modified as shown to contain this new address. Likewise, if we loaded the program beginning at address 7420 (Fig. c), the JSUB instruction would need to be changed to 4B108456 to correspond to the new address of RDREC.

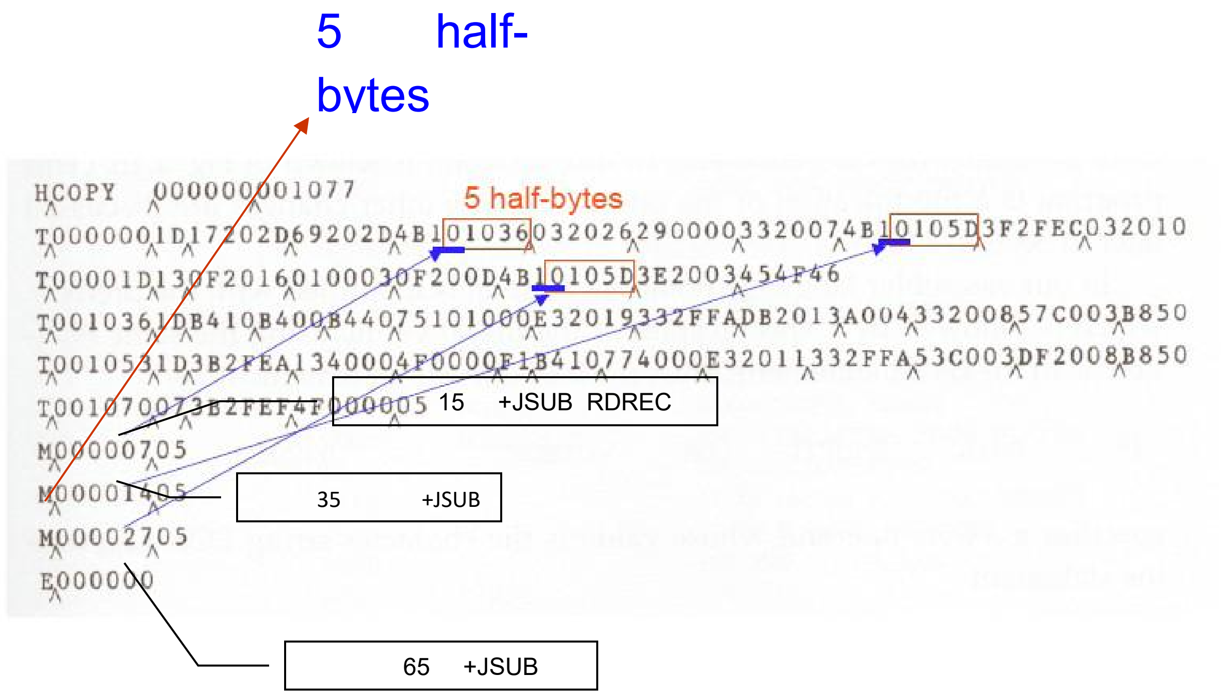
No matter where the program is loaded, RDREC always 1036 bytes past the starting address of the program. This means that we can solve the relocation problem in the following way:

3. When the assembler generated the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program.

4. The assembler will also produce a command for the loader, instructing it to add the beginning address of the program to the address field in the JSUB instruction at the load time.

The command for the loader is achieved through writing Modification Record in the Object Program.

RELOCATABLE OBJECT PROGRAM



```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

2. Briefly explain about one – pass assembler and multi-pass assembler. **

April 2013

1. Give two example program SIC operators.

i. PROGRAM TO LOAD AND STORE A NUMBER AND A CHARACTER INTO TWO VARIABLES FOR SIC

Source statement			function
	LDA	FIVE	LOAD CONSTANT 5 IN TO A REGISTER
	STA	A	STORE THE CONTENT OF THE A REGISTER IN THE VARIABLE A
	LDCH	CHARZ	LOAD CHARACTER Z INTO REGISTER A
	STCH	C	STORE THE CHARACTER FROM THE ACCUMULATER INTO THE VARIABLE C
A	RESW	1	ONE WORD VARIABLE A

FIVE	WORD	5	ONE WORD CONSTANT
CHARZ	BYTE	C ‘Z’	ONE BYTE CONSTANT
C	RESB	1	ONE BYTE VARIABLE C

ii. Program to compute the value a+b-1 and store the result in c for SIC

	LDA	A	LOAD THE VALUE OF A INTO REGISTER A
	ADD	B	ADD VALUE OF B WITH ACCUMULATOR CONTENT
	SUB	ONE	SUBTRACT ONE FROM ACCUMULATOR CONTENT
	STA	C	STORE THE VALUE a+b-1 IN C
: :			
ONE	WORD	1	ONE WORD CONSTANT
A	RESW	1	ONE WORD SPACE IS RESERVED FOR

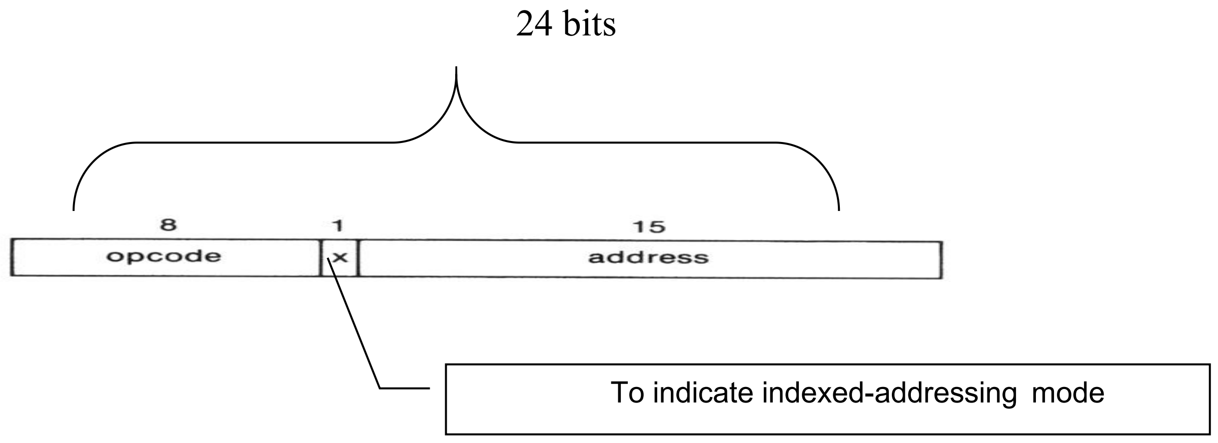
			VARIABLE A
B	RESW	1	ONE WORD VARIABLE B
C	RESW	1	ONE WORD VARIABLE C

April/May 2014

1. Explain in detail about instruction formats, instruction set and addressing modes of SIC. (11)

- **Instruction formats**:

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



Addressing modes:
Two addressing modes:

3. Direct
4. Indexed

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

( ): Contents of a register or a memory location

Target address calculation for direct addressing mode is

TA= 2001

Target address calculation for indexed addressing mode is

TA= 2000 + (X)

= 2000+1

= 2001

- **Instruction set**

  SIC provides a basic set of instructions that are sufficient for most simple tasks

  - Load and store registers
    - LDA, LDX, STA, STX
  - Integer arithmetic operations (involve register A and a word in memory, save result in A)
    - ADD, SUB, MUL, DIV
  - Comparison instruction
  - Comparison instruction

COMP instruction compares the value in the register A with another value of a variable and set the condition code CC to indicate the accumulator value is (<, =, >) the other values of a variable or word in the memory with which it is compared.

- – Conditional jump instructions (according to CC)
    - JLT (Jump Less Than), JEQ(Jump Equal To), JGT(Jump Greater Than) instruction test the setting of CC and jumps accordingly.
- Eg.

```
                :
                :
                COMP   B { compare value of B with the accumulkator
                JLT       WLOOP {if accumulator content is < B jump to WLOOP}
WLOOP                  ADD      INCR { Add the value of INCR to the accumulator}
```

- – Subroutine linkage
    - JSUB (jumps and places the return address in register L)
    - RSUB (returns to the address in L)

2. Explain briefly about machine independent assembler features? (11)**

LANGUAGE TRANSLATORS

QUESTION BANK WITH ANSWERS

Note: * - Repeated Questions

UNIT: 2

PART A TWO MARKS

1. What is address sensitive program?

   Assembly language program **(ALP**) is address sensitive program.

2. List out the function performed by relocating loaders?

   Loader that allow program location are called relocation loaders

   Two methods for specifying relocation loaders

   Relocation by modification record method

   Relocation by bit mask method

3. What is Bootstrap Loader?

   This is a special type of absolute loader which loads the first program to be run by the computer. (Usually an operating system)

4. What is mean by OPTAB, SYMTAB?

   Assembler uses two major data structure

   1. OPTAB (Operation code table)

   Use: Look up mnemonic operation codes and translate them to their machine language equivalents

$$\begin{tikzcd}
 & \text{Mnemonic opcode} \\
\text{Fields in OPTAB} \arrow[ur] \arrow[dr] & \\
 & \text{Machine language equivalent}
\end{tikzcd}$$

   2. SYMTAB (Symbol table)

   Use: Store the addresses of labels

$$\begin{tikzcd}
 & \text{Name of symbol} \\
\text{Fields in SYMTAB} \arrow[ur] \arrow[r] \arrow[dr] & \text{Address} \\
 & {}
\end{tikzcd}$$

5. List the assembler director.
   - EQU
     syntax

     Symbol EQU value

     It establishes symbolic names that can be used for improving the readability of the statement.
   - ORG

     ORG value

     When this statement is encountered the assembler resets its location counter to the specified value.

6. What is mean by OPTAB, SYMTAB? **

7. What is linking required after a program is translated? Define link editor.
   Linking function is performed at execution time. The operating system is used to load and link subprograms at the time they are first called.
   Linkage editor performs relocation of all controls sections relative to the start of the linked program and loading is accomplished in one pass with no external table requirement.

8. What is an absolute loader? State its advantages.

   Absolute loader does not need to perform functions as linking and relocation its operation is very simple.

   For a simple absolute loader, all functions are accomplished in a single pass as follows:
   1) The Header record of object programs is checked to verify that the correct program has been presented for loading.
   2) As each Text record is read, the object code it contains is moved to the indicated address in memory.
   3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program

9. What are the uses of bootstrap loader?

   When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.
   In PC It is used to boot he OS of the system. In case of *microcontrollers*, **a boot loader** enriches the capabilities of the microcontroller and makes them self programmable device**.**

10. Differentiate near and far jump?

Near jump

A near jump is a jump to a target in the same segment and it is assembled by using a current code segment CS.

Far jump

A far jump is a jump to a target in a different code segment and it is assembled by using different segment registers

PART B (11 MARK)

1. Discuss the functions and design of an absolute loader. (11)

Design of an Absolute Loader

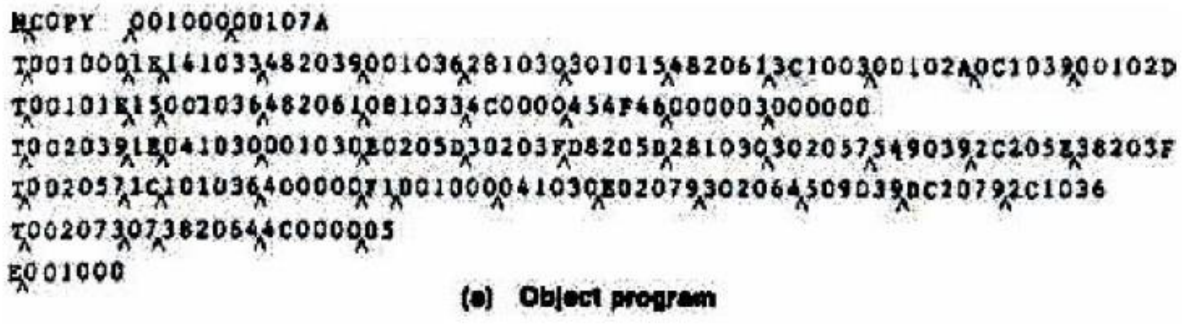
For a simple absolute loader, all functions are accomplished in a single pass as follows:

1) The Header record of object programs is checked to verify that the correct program has been presented for loading.

2) As each Text record is read, the object code it contains is moved to the indicated address in memory.

3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

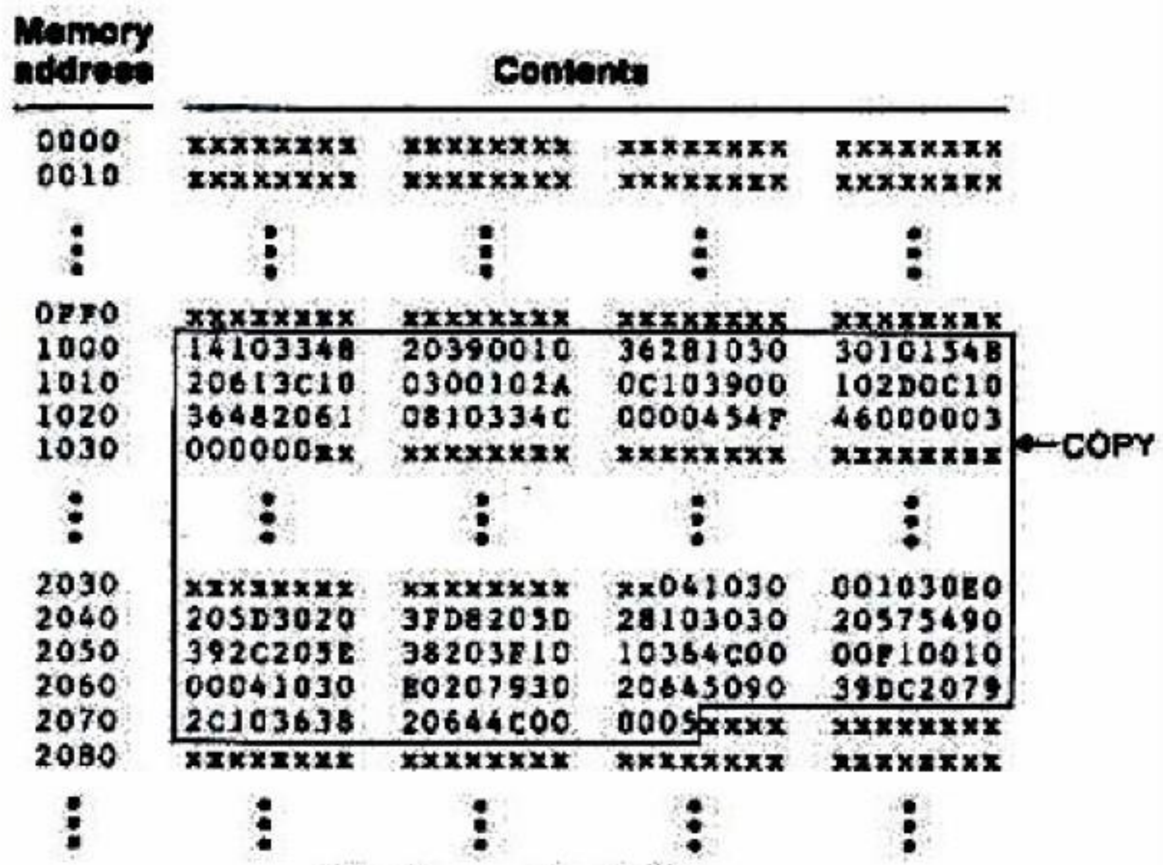
An example object program is shown in Fig (a).



```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

Fig (b) shows a representation of the program from Fig (a) after loading.



```
Memory
address      Contents

0000      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
0010      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  :          :         :         :         :
0FF0      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
1000      14103348  20390010  36281030  3010154B
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

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
```

- ➢ It is very important to realize that in Fig (a), each printed character represents one byte of the object program record.
- ➢ In Fig (b), on the other hand, each printed character represents one hexadecimal digit in memory (a half-byte).
- ➢ Therefore, to save space and execution time of loaders, most machines store object programs in a **binary form**, with each byte of object code stored as a single byte in the object program.
- ➢ In this type of representation a byte may contain any binary value.

2. a. What is dyanamic linking? Explain. (6)*

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.

Dynamic Linking Applications

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.
- For example, a single copy of the standard C library can be loaded into memory.
- All C programs currently in execution can be linked to this one copy, instead of linking a separate copy into each object program.
- In an object-oriented system, dynamic linking is often used for references to software object.

- This allows the implementation of the object and its method to be determined at the time the program is run. (e.g., C++)
- The implementation can be changed at any time, without affecting the program that makes use of the object.

Dynamic Linking Advantage

- The subroutines that diagnose errors may never need to be called at all.
- However, without using dynamic linking, these subroutines must be loaded and linked every time the program is run.
- Using dynamic linking can save both space for storing the object program on disk and in memory, and time for loading the bigger object program.

➢ Linkage editors perform linking operations before the program is loaded for execution.

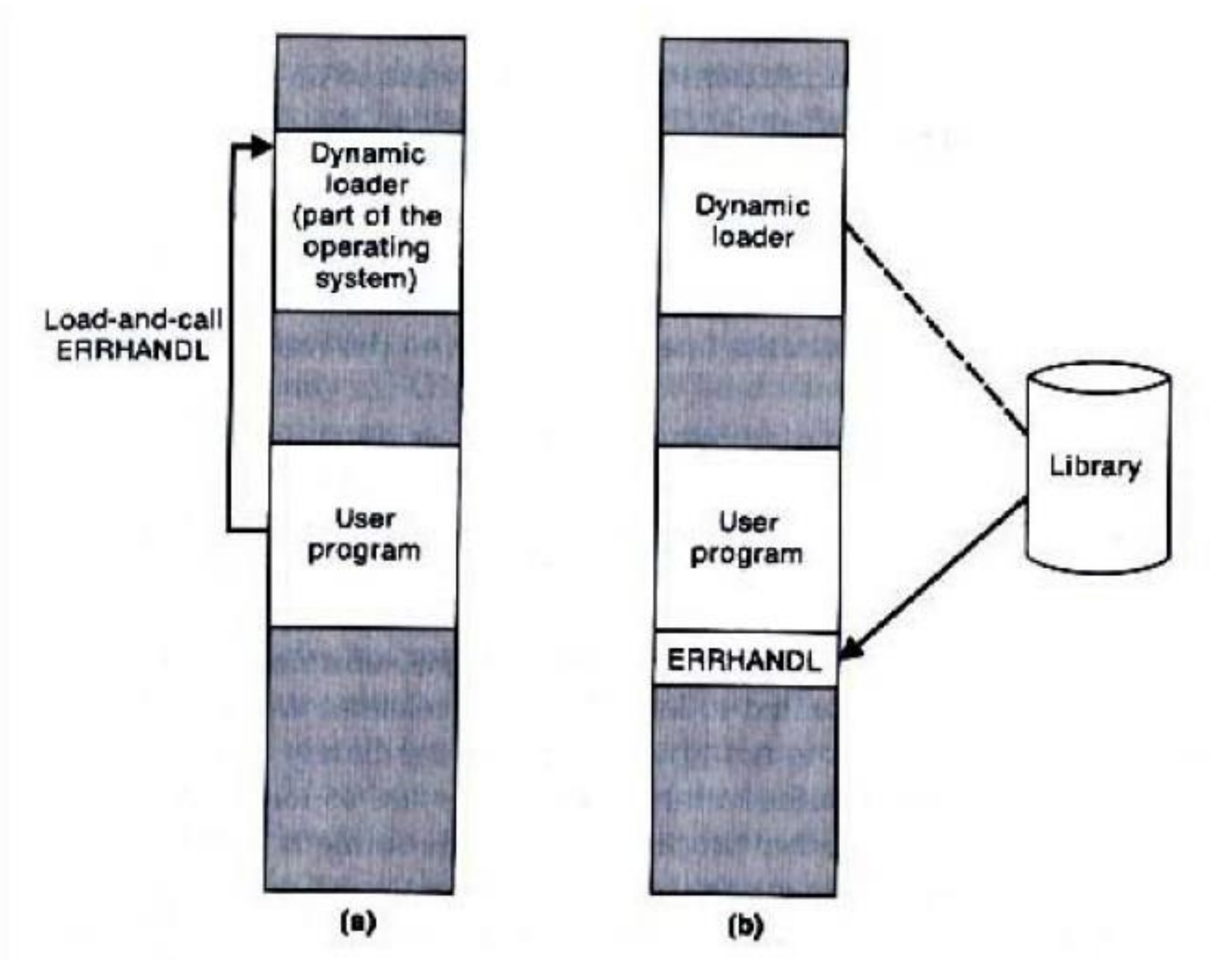
➢ Linking loaders perform these same operations at load time.

➢ Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.

➢ Dynamic linking is often used to allow several executing programs to share one of a subroutine or library, ex. run-time support routines for a high-level language like C.

➢ With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.

➢ Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.


Dynamic loader (part of the operating system)
Load-and-call ERRHANDL
User program
(a)
Dynamic loader
Library
User program
ERRHANDL
(b)

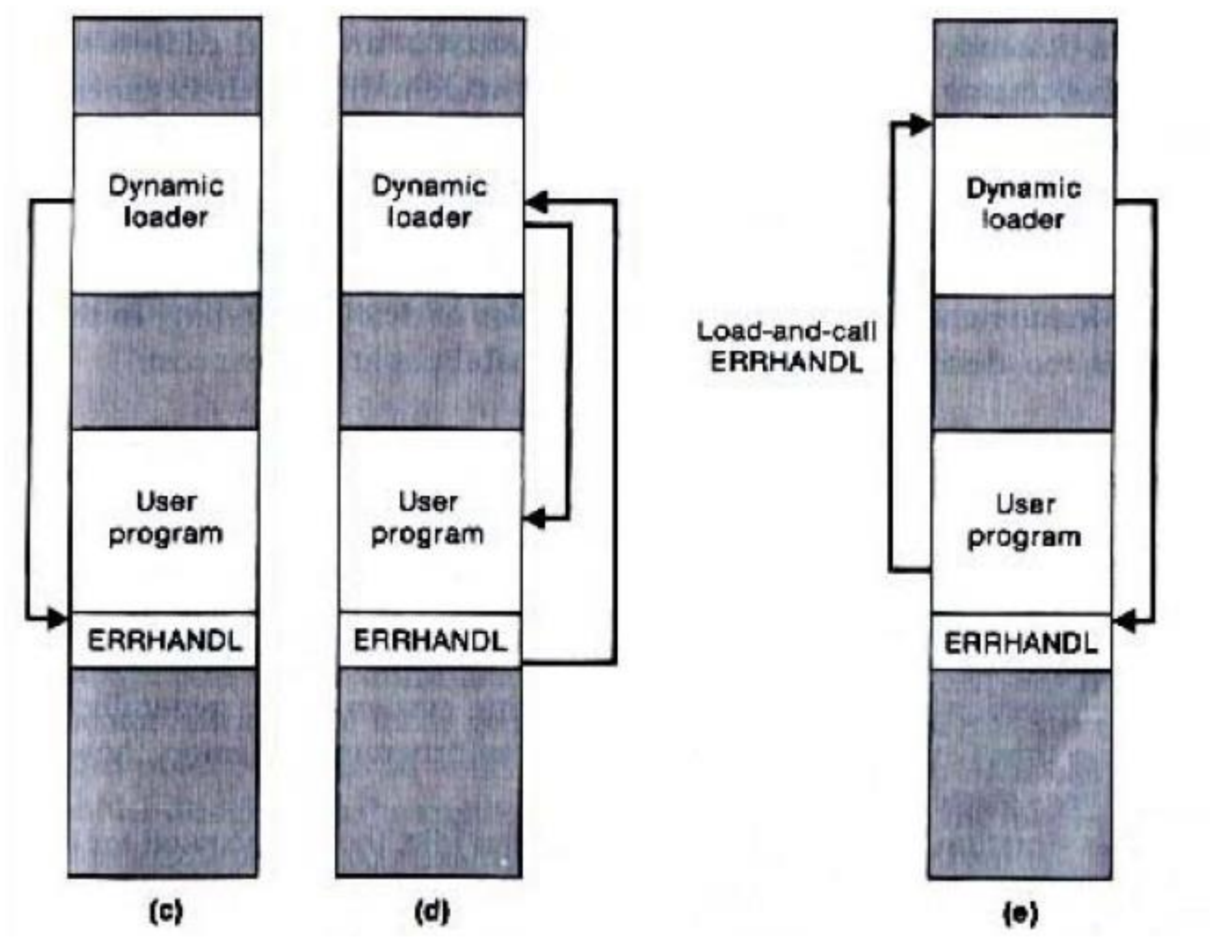


Fig (a): Instead of executing a JSUB instruction referring to an external symbol, the program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.
Fig (b): OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.
Fig (c): Control is then passed from OS to the routine being called **Fig (d):** When the called subroutine completes it processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.
Fig (e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.

b. Write about bootstrap loaders. (5)

Bootstrap Loaders

- With the machine empty and idle there is no need for program relocation.
- We can specify the absolute address for whatever program is first loaded and this will be the OS, which occupies a predefined location in memory.
- We need some means of accomplishing the functions of an absolute loader.
  1. To have the operator enter into memory the object code for an absolute loader, using switches on the computer console.
  2. To have the absolute loader program permanently resident in a ROM.
  3. To have a built –in hardware function that reads a fixed –length record from some device into memory at a fixed location.
- When some hardware signal occurs, the machine begins to execute this ROM program.
- On some computers, the program is executed directly in the ROM: on others, the program is copied from ROM to main memory and executed there.
- The particular device to be used can often be selected via console switches.
- After the read operation is complete, control is automatically transferred to the address in memory where the record was stored, which contains machine where the record was stored, which contains machine instructions that load the absolute program that follow.
- If the loading process requires more instructions that can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of still more records – boots trap.
  The first record is generally referred to as bootstrap loader:
- Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.
- This includes the OS itself and all stand-alone programs that are to be run without an OS.

3. Discuss about basic loader functions.

BASIC LOADER FUNCTIONS

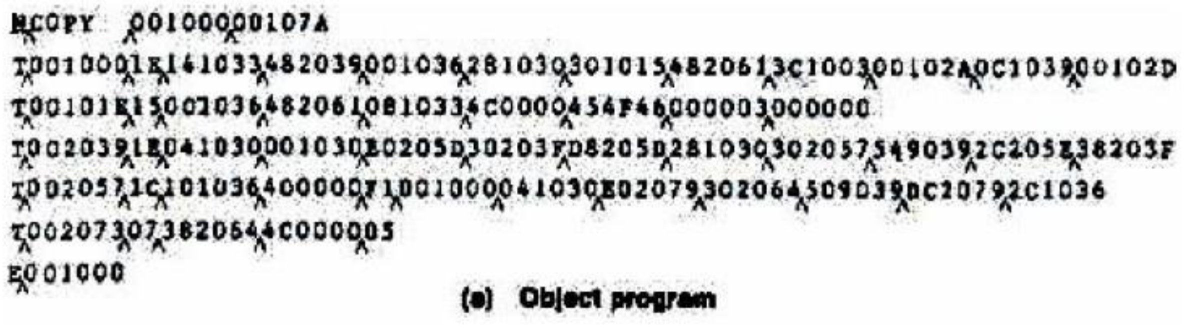
Fundamental functions of a loader:
1. Bringing an object program into memory.
2. Starting its execution.

3.1.1 Design of an Absolute Loader

For a simple absolute loader, all functions are accomplished in a single pass as follows:
1) The Header record of object programs is checked to verify that the correct program has been presented for loading.
2) As each Text record is read, the object code it contains is moved to the indicated address in memory.

3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

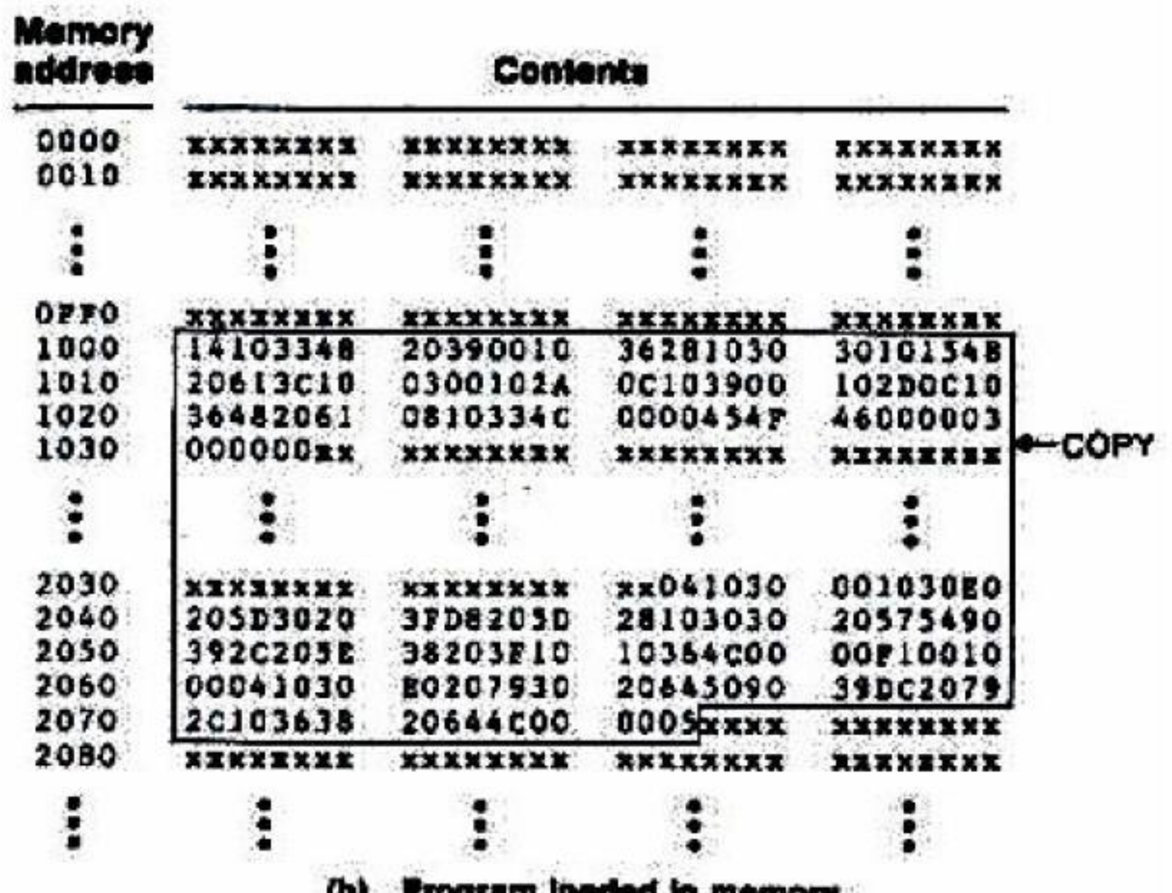
An example object program is shown in Fig (a).



```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

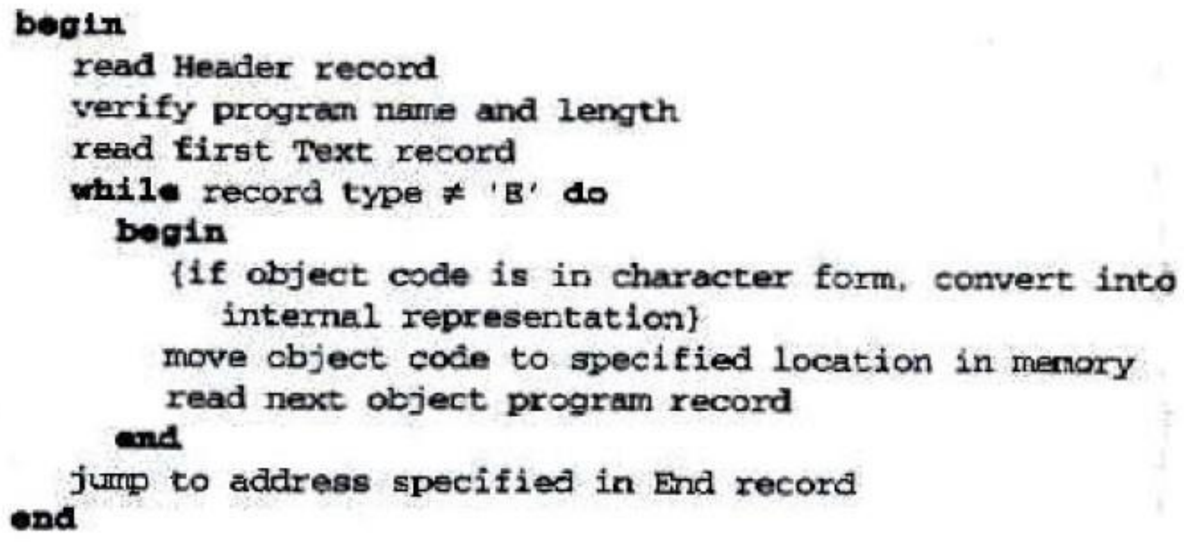
Fig (b) shows a representation of the program from Fig (a) after loading.



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

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
```

- ➢ It is very important to realize that in Fig (a), each printed character represents one byte of the object program record.
- ➢ In Fig (b), on the other hand, each printed character represents one hexadecimal digit in memory (a half-byte).
- ➢ Therefore, to save space and execution time of loaders, most machines store object programs in a **binary form**, with each byte of object code stored as a single byte in the object program.
- ➢ In this type of representation a byte may contain any binary value.

A Simple Bootstrap Loader

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

Working of a simple Bootstrap loader

- ➢ The bootstrap begins at address 0 in the memory of the machine.
- ➢ It loads the operating system at address 80.
- ➢ Each byte of object code to be loaded is represented on device F1 as *two hexadecimal digits* just as it is in a Text record of a SIC object program.

- ➢ The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.
- ➢ 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.
- ➢ Much of the work of the bootstrap loader is performed by the subroutine GETC. GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object

code to be loaded. For example, two bytes = C “D8” ‘4438’H converting to one byte ‘D8’H.

- The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.
- The TIXR instruction is then used to add 1 to the value in X.

4. Explain the machine independent loader features.

MACHINE-INDEPENDENT LOADER FEATURES

- Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler.
- They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- Linking loaders that support *automatic library search* must keep track of external 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 or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search 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 therefore necessary to repeat the library search process until all references are resolved.
- If unresolved external references remain after the library search is completed, these must be treated as errors.

Loader Options

Many loaders allow the user to specify options that modify the standard processing

Typical loader option 1: Allows the selection of alternative sources of input.

Ex : INCLUDE program-name (library-name) might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

Loader option 2: Allows the user to delete external symbols or entire control sections.

Ex : DELETE csect-name might instruct the loader to delete the named control section(s) 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 programs.

Loader option 3: Involves the automatic inclusion of library routines to satisfy external references.

Ex. : LIBRARY MYLIB

Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.
NOCALL STDDEV, PLOT, CORREL

- To instruct the loader that these external references are to unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

LOADER DESIGN OPTIONS

Linking loaders perform all linking and relocation at load time.
There are two alternatives:
1. **Linkage editors**, which perform linking prior to load time.
2. **Dynamic linking**, in which the linking function is performed at execution time.
   - Precondition: The source program is first assembled or compiled, producing an object program.
   - A **linking loader** performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.
   - A **linkage editor** produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

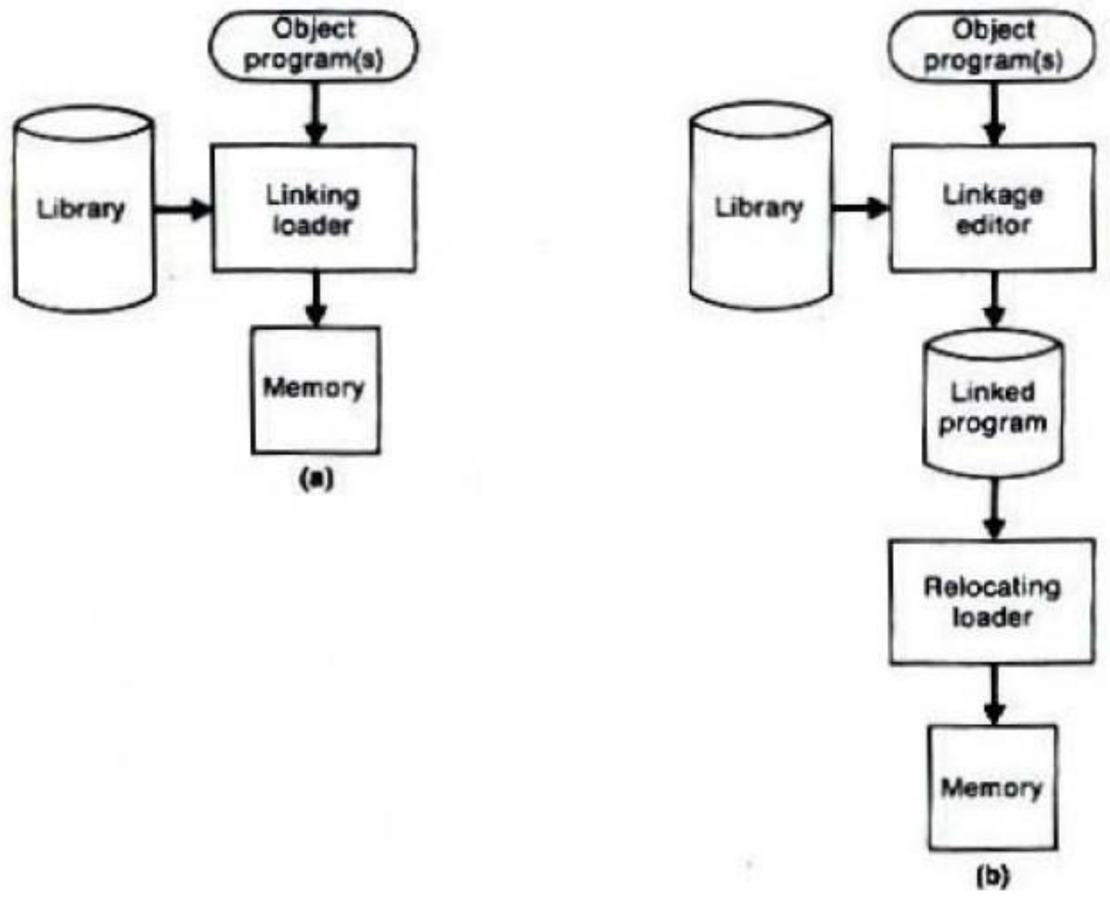
5. Discuss about machine independent loader features. *****
6. Explain the different options of loader design.
   1. LINKAGE EDITOR
   2. DYNAMIC LINKING **********
   3. BOOTSTAP LOADER*************

1. Linkage Editors

- The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, 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 an inkage editor substantially reduces the overhead required.Linkage editors can perform any useful functions besides simply preparing an object program for execution. Ex., a typical sequence of linkage editor commands used:
  INCLUDE PLANNER (PROGLIB)
  DELETE PROJECT **{delete from existing PLANNER}**
  INCLUDE PROJECT (NEWLIB) **{include new version}**
  REPLACE PLANNER (PROGLIB)

- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages.
- Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

Fig (7): Processing of an object program using (a) Linking loader and (b) Linkage Editor



7. Define the bootstrap loader & state the algoritm for the same. **

8. Explain with neat block diagram the role of loader & linker.**

9. Write briefly about relocation techniques (11)

MACHINE-DEPENDENT LOADER FEATURES

- The absolute loader has several potential disadvantages. One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory.
- On a simple computer with a small memory the actual address at which the program will be loaded can be specified easily.
- On a larger and more advanced machine, we often like to run several independent programstogether, sharing memory between them. We do not know in advance where a program will be loaded. Hence we write relocatable programs instead of absolute ones.
- Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all of the subroutines had preassigned absolute addresses.
- The need for *program relocation* is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics.
- Loaders that allow for program relocation are called relocating loaders or relative loaders.

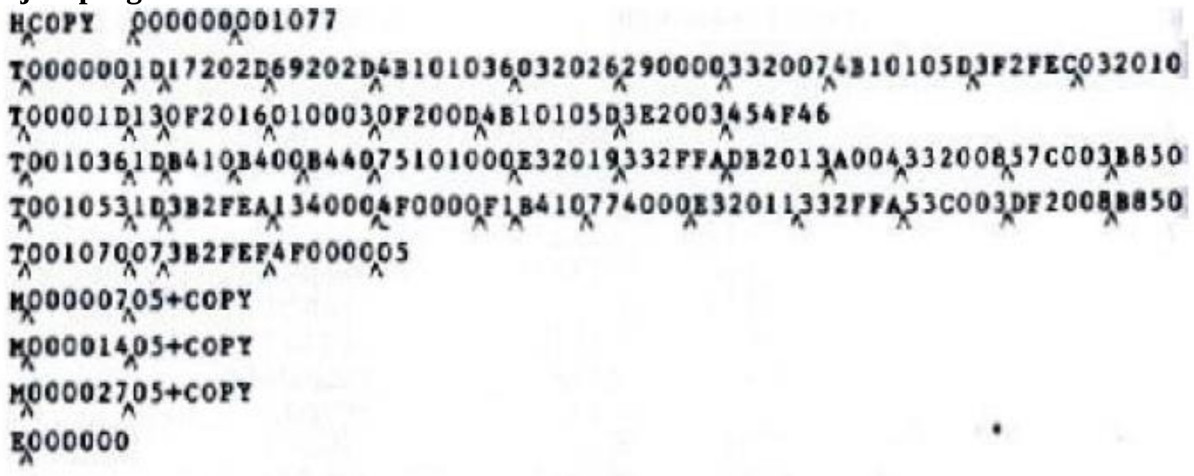
Relocation

Two methods for specifying relocation as part of the object program:

The first method:

- A Modification 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 xtended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

Object program

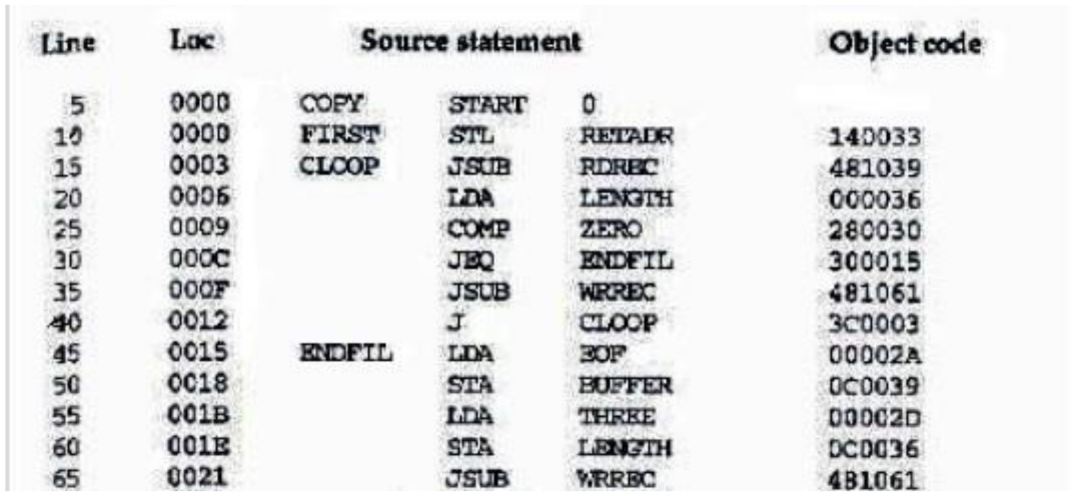


```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05+COPY
M^000014^05+COPY
M^000027^05+COPY
E^000000
```

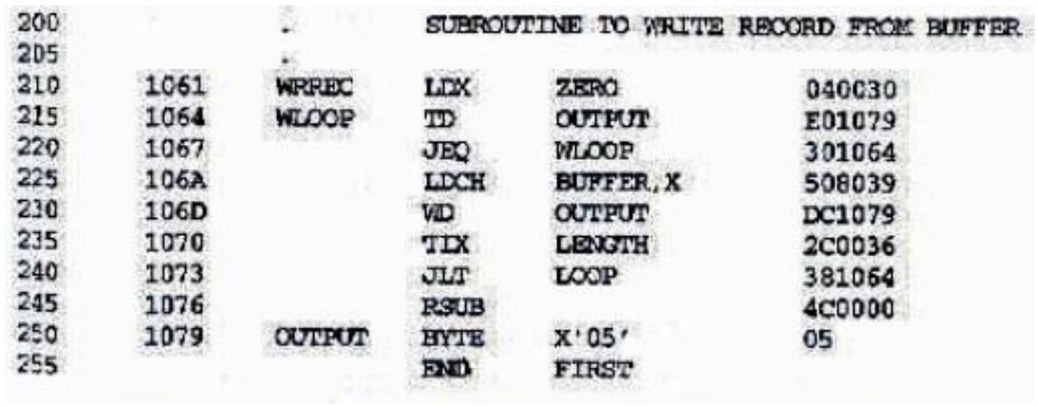
- Each Modification record specifies the starting address and length of the field whose value is to be altered.
- It then describes the modification to be performed.
- In this example, all modifications add the value of the symbol COPY, which

represents the starting address of the program.

Fig(2) :Consider a Relocatable program for a Standard SIC machine



Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061



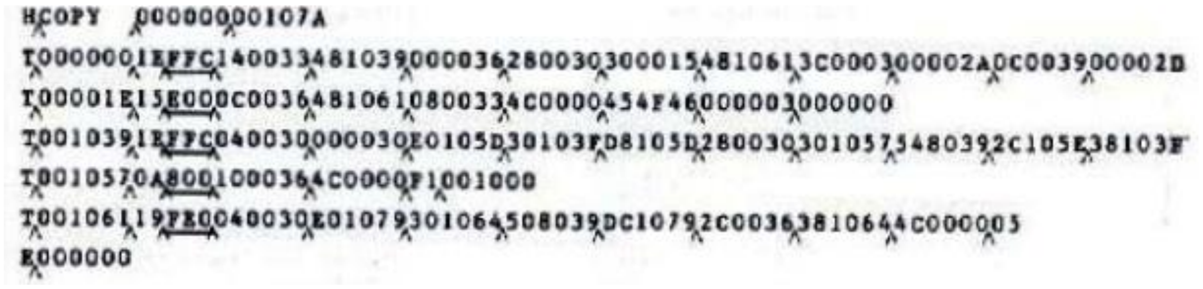
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	1061	WRREC	LDX	ZERO	040030
215	1064	WLOOP	TD	OUTPUT	E01079
220	1067		JEQ	WLOOP	301064
225	106A		LDCH	BUFFER,X	508039
230	106D		WD	OUTPUT	DC1079
235	1070		TIX	LENGTH	2C0036
240	1073		JLT	LOOP	381064
245	1076		RSUB		4C0000
250	1079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

- The Modification record is not well suited for use with all machine architectures.Consider, for example, the program in Fig (2) .This is a relocatable program written for standard version for SIC.
- The important difference between this example and the one in Fig (1) is that the standard SIC machine does not use relative addressing.
- In this program the addresses in all the instructions except RSUB must modified when the program is relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig (1).

The second method:

- There are no Modification records.
- The Text records are the same as before except that there is a *relocation bit* associated with each word of object code.
- Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.

Fig (3): Object program with relocation by bit mask



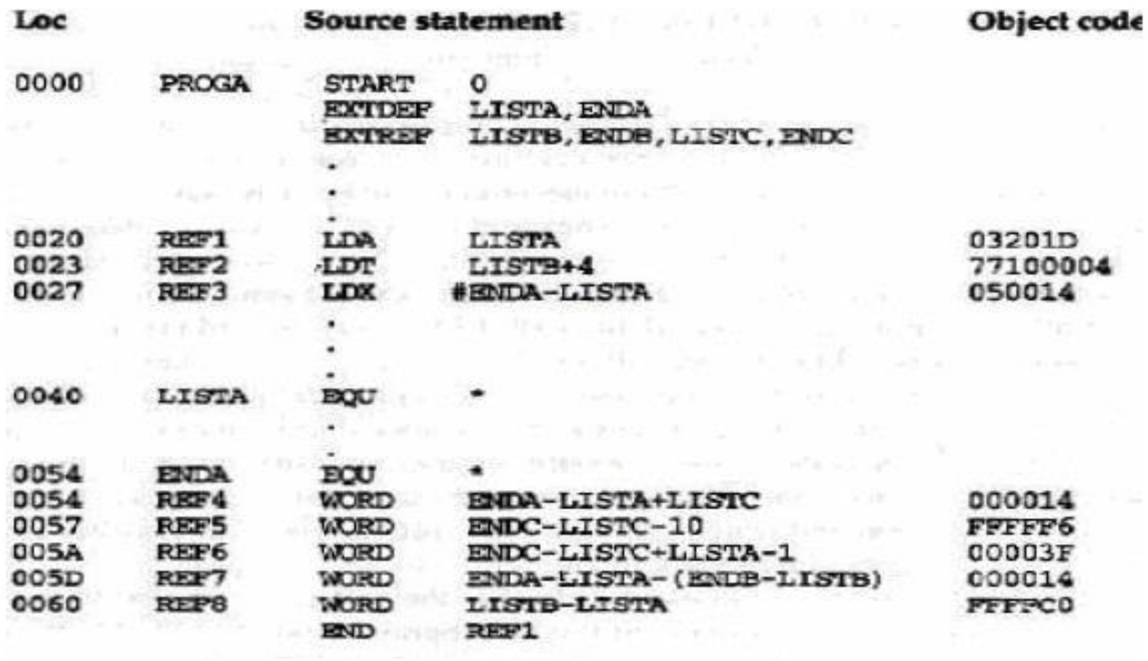
```
HCOPY  00000000107A
T0000001EFFC1400334810390000362800303000154810613C000300002A0C003900002D
T00001E15E000C00364810610800334C0000454F46000003000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C0036381064 4C000005
E000000
```

- The relocation bits are gathered together into a **bit mask** following the length indicator in each Text record. In Fig (3) this mask is represented (in character form) as three hexadecimal digits.
- If 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. A bit value of **0** indicates that no modification is necessary.

- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0. For example, the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

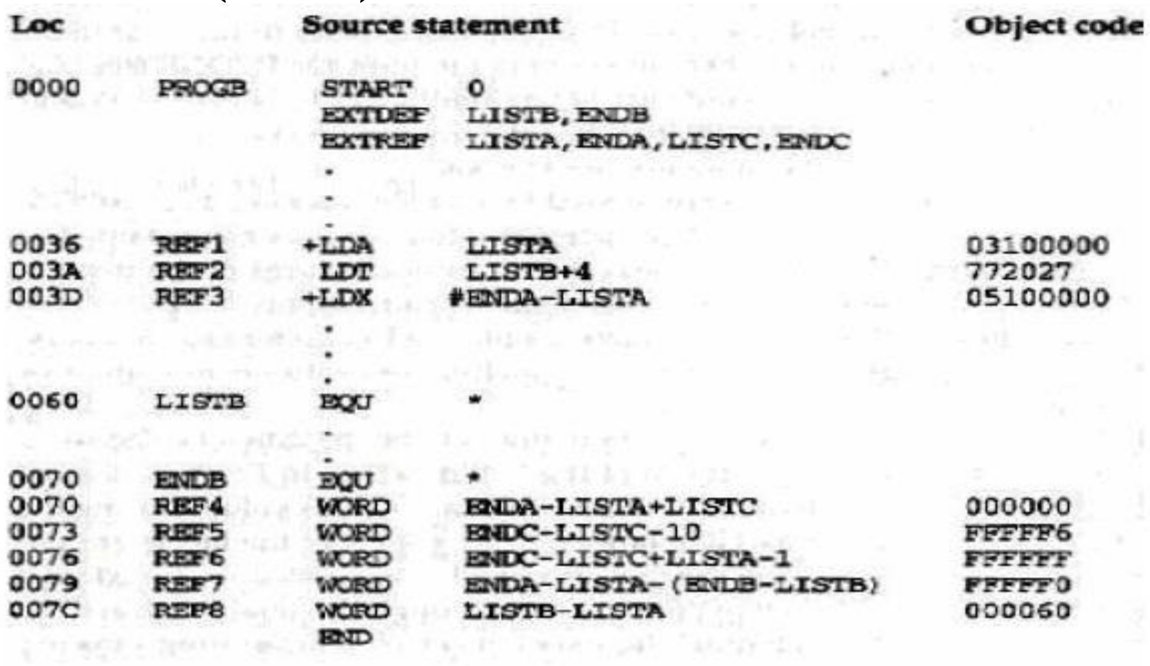
Example: Note that the LDX instruction on line 210 (Fig (2)) begins a new Text record. If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185.

Program 1 (PROGA):



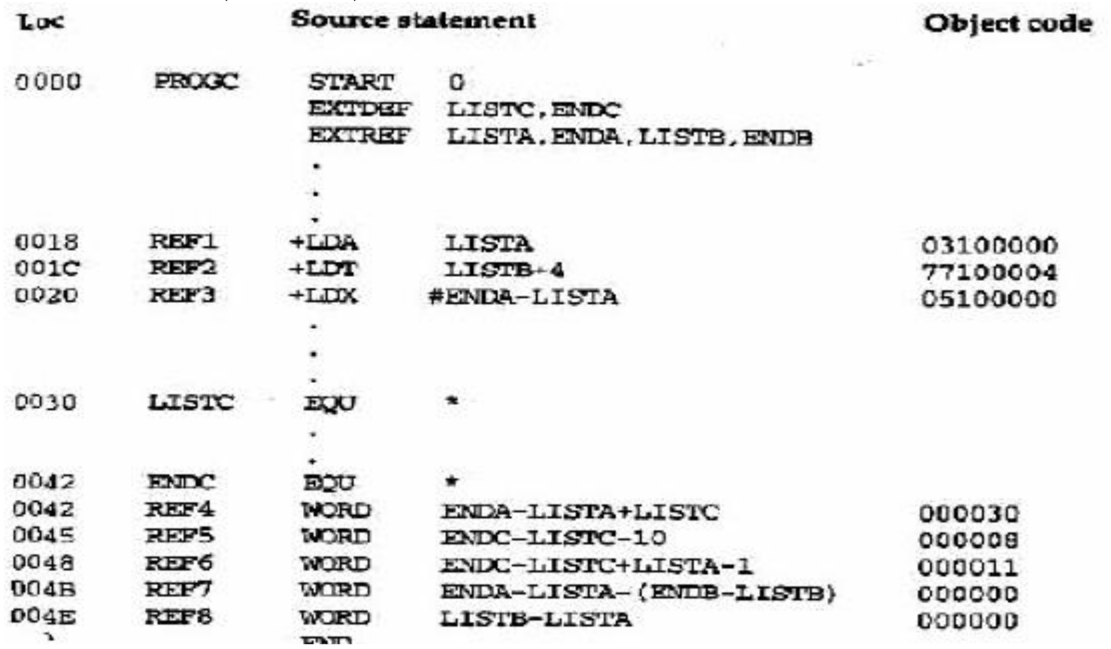
```
Loc                 Source statement                        Object code

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 2(PROGB)



```
Loc              Source statement                                Object code

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
```

PROGRAM3 (PROGC)



```
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
```

Consider first the reference marked REF1.

For the first program (PROGA),

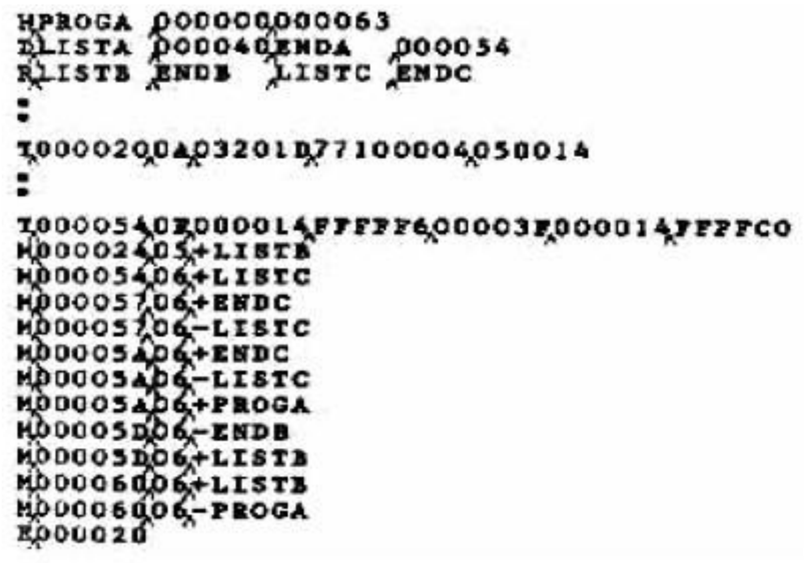
- REF1 is simply a reference to a label within the program.
- It is assembled in the usual way as a PC relative instruction.

- No modification for relocation or linking is necessary.
- In PROGB, the same operand refers to an external symbol.
- The assembler uses an extended-format instruction with address field set to 00000.
- The object program for PROGB contains a Modification record instructing the loader to add *the value of the symbol LISTA* to *this address field* when the program is linked.

For PROGC, REF1 is handled in exactly the same way.

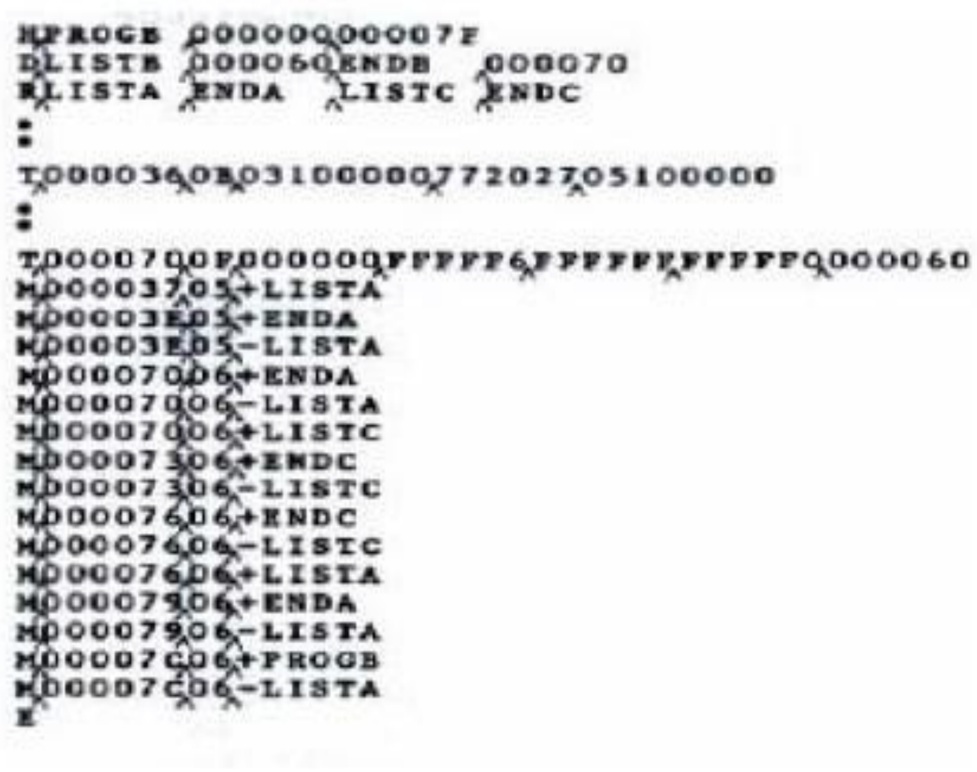
Corresponding object programs

PROGA



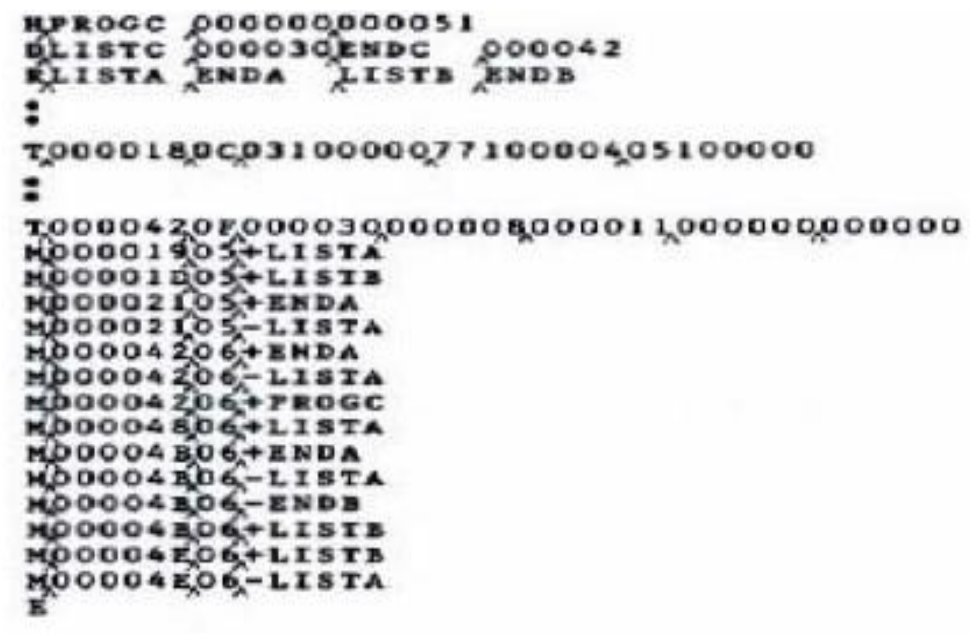
```
HPROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
:
T^000020^0A^03201D^77100004^050014
:
T^000054^0F^000014^FFFFF6^00003F^000014^FFFFC0
M^000024^05^+LISTB
M^000054^06^+LISTC
M^000057^06^+ENDC
M^000057^06^-LISTC
M^00005A^06^+ENDC
M^00005A^06^-LISTC
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB
M^000060^06^+LISTB
M^000060^06^-PROGA
E^000020
```

PROGB



```
H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB  ^000070
R^LISTA ^ENDA  ^LISTC ^ENDC
:
T^000036^0B^03100000^772027^05100000
:
T^000070^0F^000000^FFFFF6^FFFFFF^FFFFF0^000060
M^000037^05^+LISTA
M^00003E^05^+ENDA
M^00003E^05^-LISTA
M^000070^06^+ENDA
M^000070^06^-LISTA
M^000070^06^+LISTC
M^000073^06^+ENDC
M^000073^06^-LISTC
M^000076^06^+ENDC
M^000076^06^-LISTC
M^000076^06^+LISTA
M^000079^06^+ENDA
M^000079^06^-LISTA
M^00007C^06^+PROGB
M^00007C^06^-LISTA
E
```

PROGC



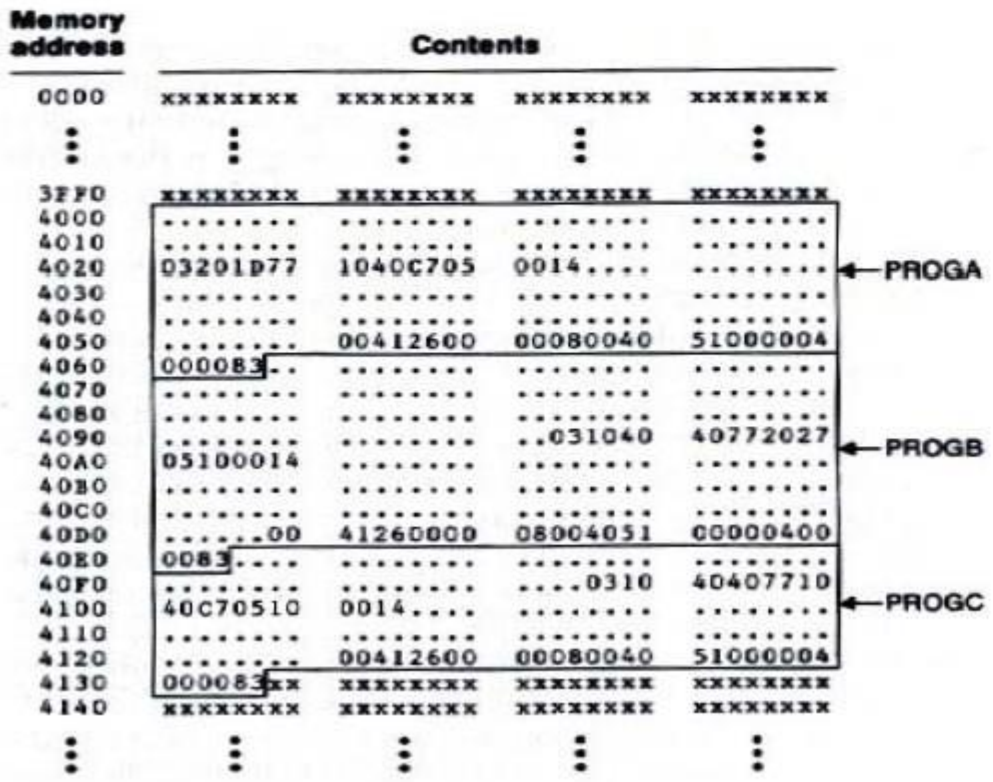
```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
:
T0000180C03100000077100004051000000
:
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

- ➢ The reference marked REF2 is processed in a similar manner.
- ➢ REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes).
- ➢ In PROGA, the assembler has all of the information necessary to compute this value. During the assembly of PROGB (and PROGC), the values of the labels are unknown. In these programs, the expression must be assembled as an external reference (*with two Modification records*) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

- **Consider REF4.**

- ➢ The assembler for PROGA can evaluate all of the expression in REF4 except for the value of LISTC. This results in an initial value of ‘000014’H and one Modification record.
- ➢ The same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and *three* Modification records.
- ➢ For PROGC, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded).
- ➢ The initial value of this data word contains the relative address of LISTC (‘000030’H). Modification records instruct the loader to add the beginning address of the program (i.e., the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA.

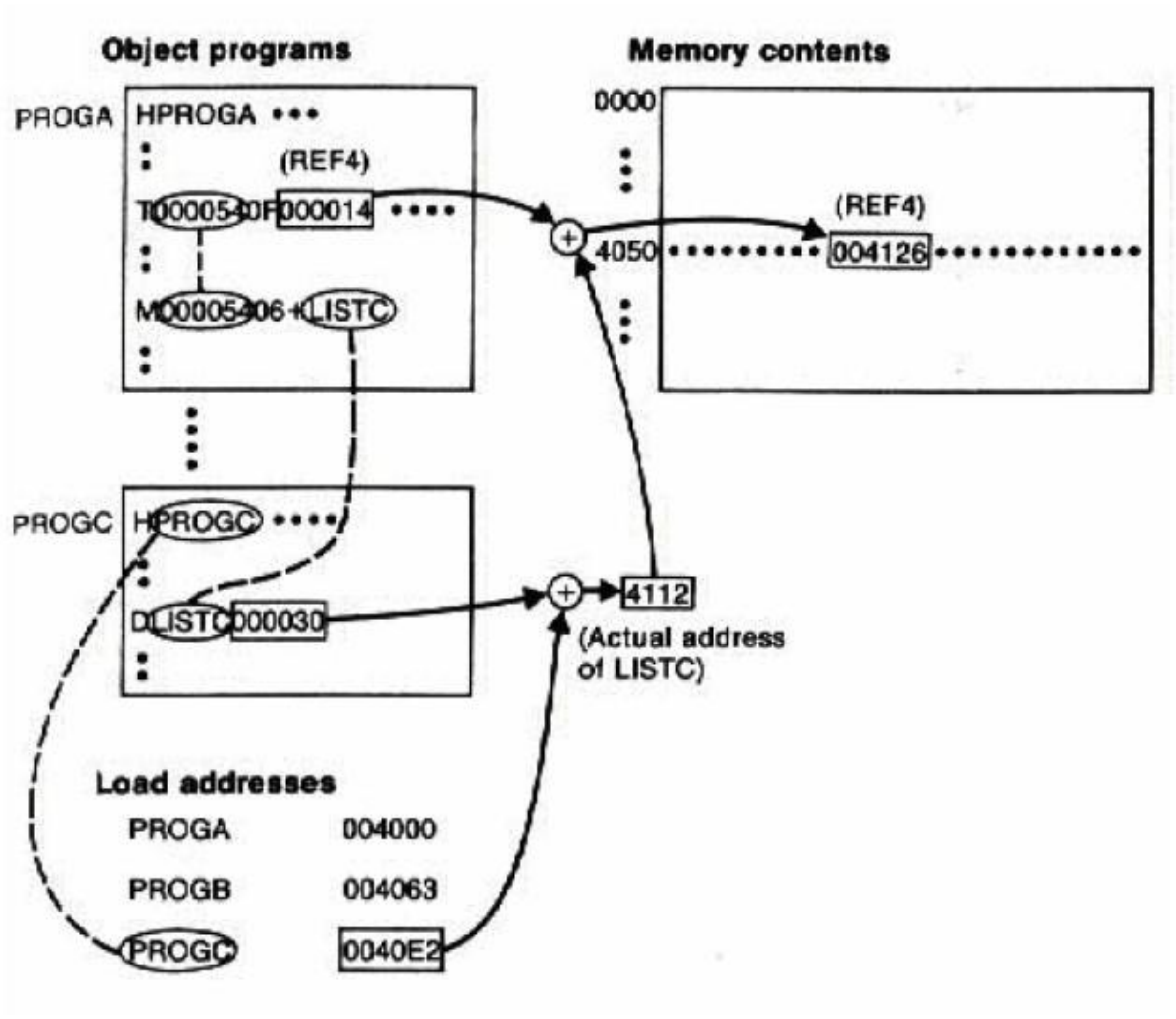
Fig (4): The three programs as they might appear in memory after loading and linking.



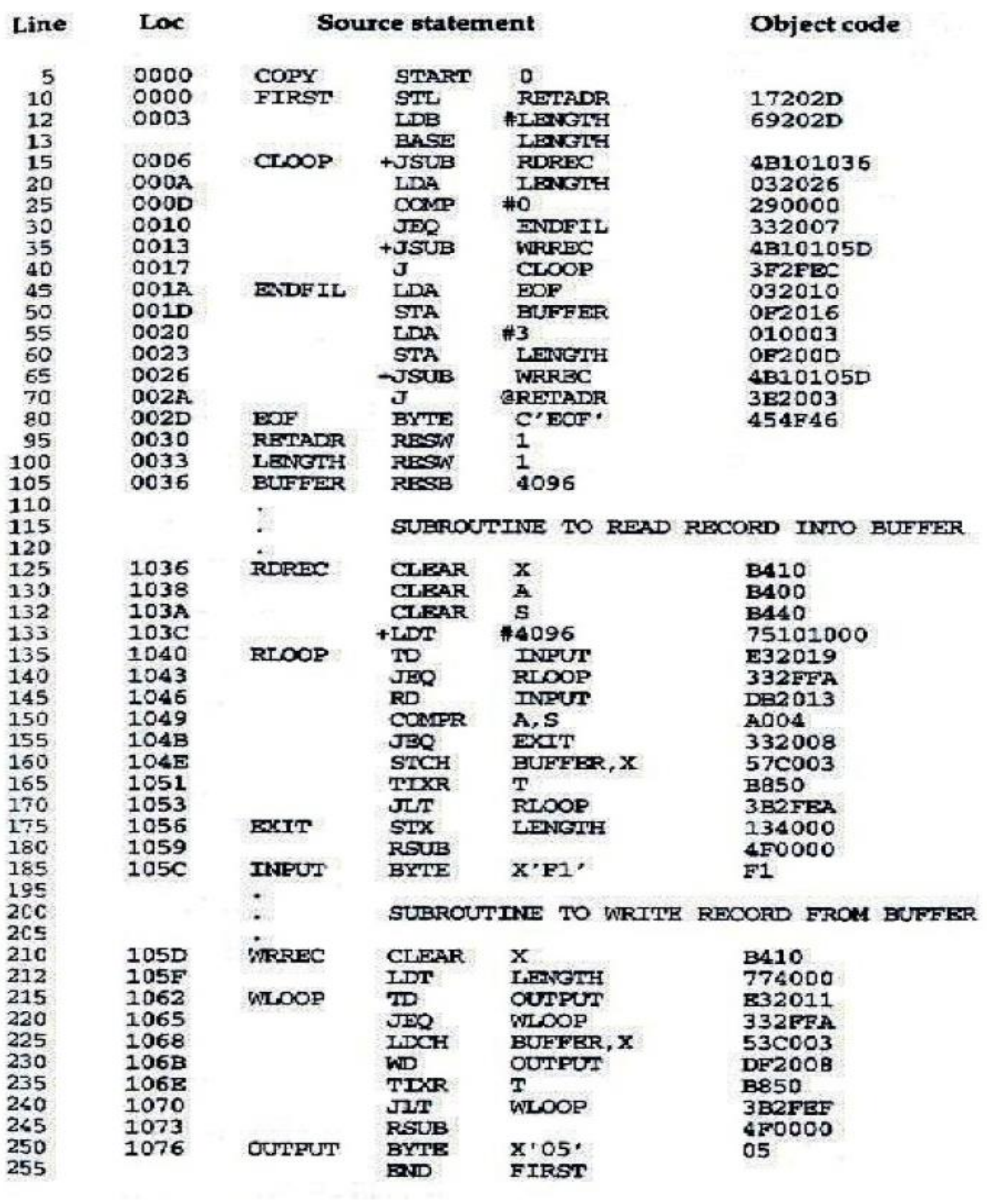
```
Memory
address              Contents

0000     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  :         :         :         :         :
3FF0     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
4000     ........  ........  ........  ........
4010     ........  ........  ........  ........
4020     03201D77  1040C705  0014....  ........  ←PROGA
4030     ........  ........  ........  ........
4040     ........  ........  ........  ........
4050     ........  00412600  00080040  51000004
4060     000083..  ........  ........  ........
4070     ........  ........  ........  ........
4080     ........  ........  ........  ........
4090     ........  ........  ..031040  40772027  ←PROGB
40A0     05100014  ........  ........  ........
40B0     ........  ........  ........  ........
40C0     ........  ........  ........  ........
40D0     ......00  41260000  08004051  00000400
40E0     0083....  ........  ........  ........
40F0     ........  ........  ....0310  40407710
4100     40C70510  0014....  ........  ........  ←PROGC
4110     ........  ........  ........  ........
4120     ........  00412600  00080040  51000004
4130     000083xx  xxxxxxxx  xxxxxxxx  xxxxxxxx
4140     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  :         :         :         :         :
```

PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.
For example, the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054).

Fig (5): Relocation and linking operations performed on REF4 in PROGA



The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which 4112 (the beginning address of PROGC plus 30).



Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

10. Explain the operation of dynamic linking (11)

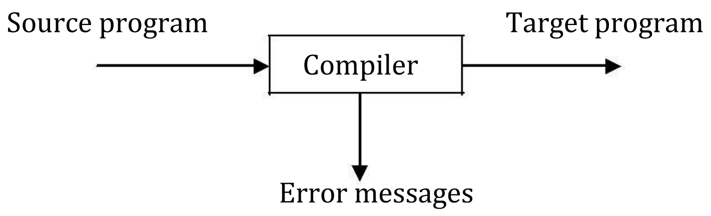
UNIT –III

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.

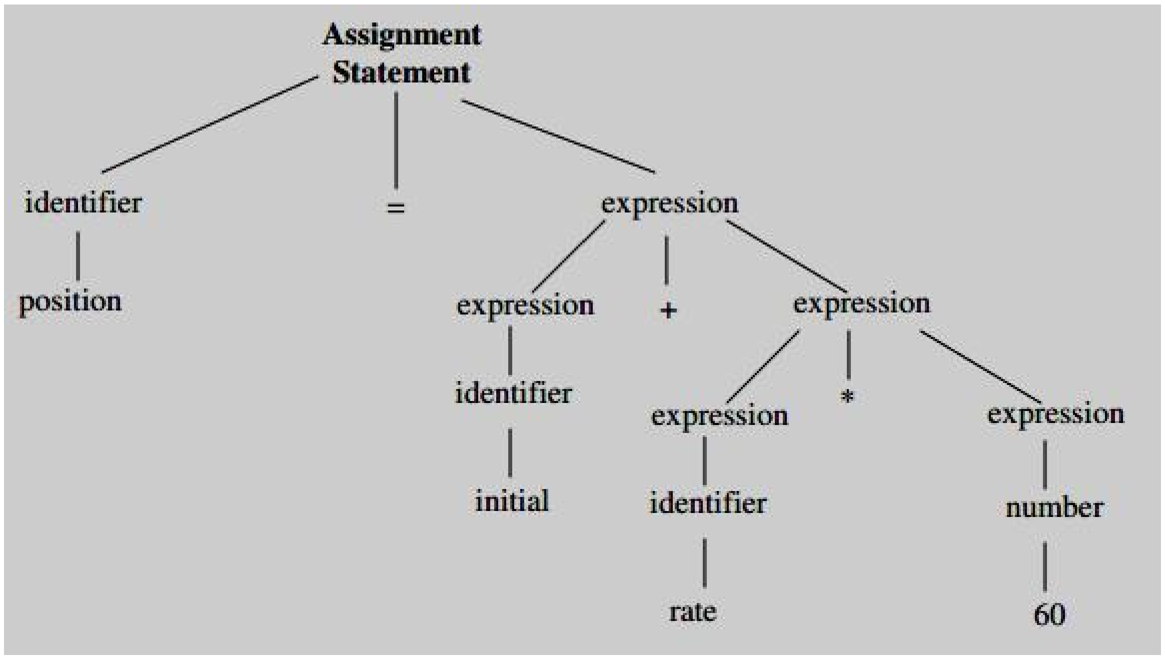
8. What is hierarchical analysis? (NOV 2011)

- *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?

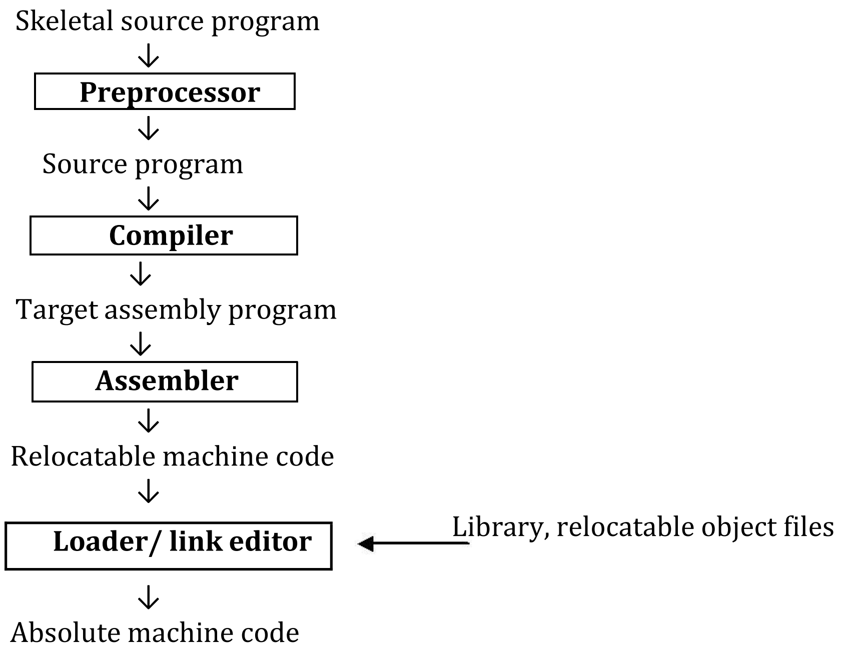
- The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions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

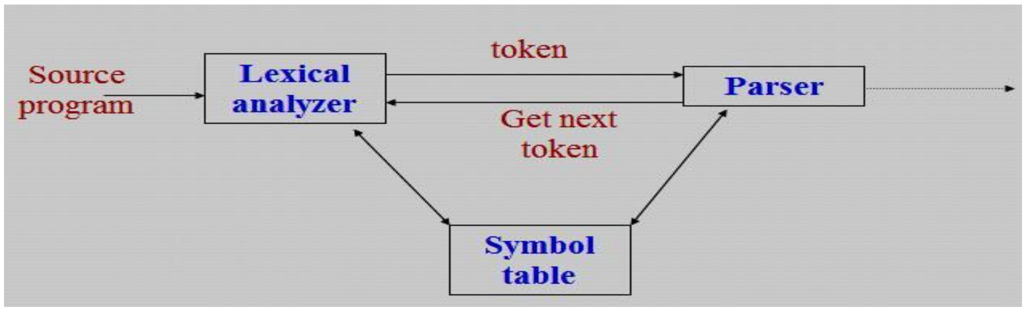
26. List the various compiler construction tools.

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.

ε is a regular expression that denotes {ε}, the set containing empty string.

2. If *a* is a symbol in Σ, 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

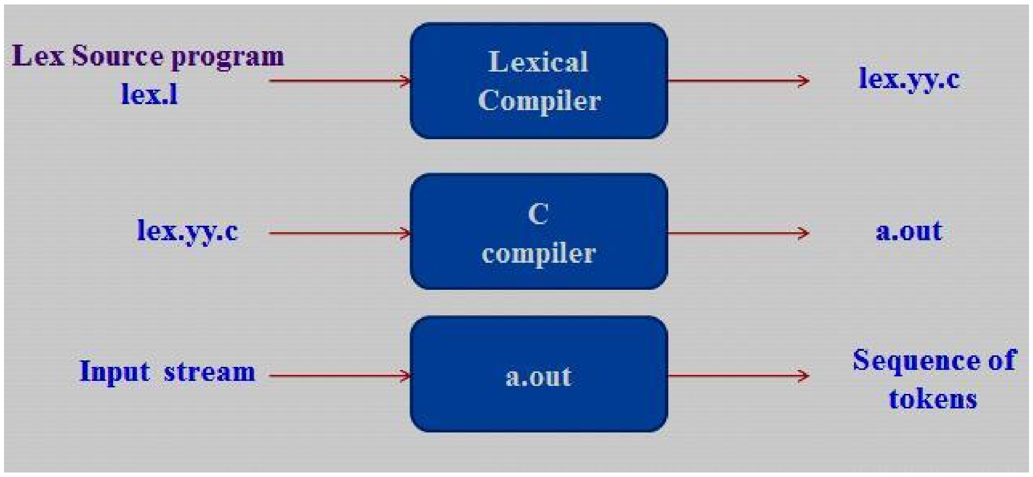
41. What is transition diagram? (NOV 2012)

- 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 { $action_1$}

p_2 { $action_2$}

…… …..

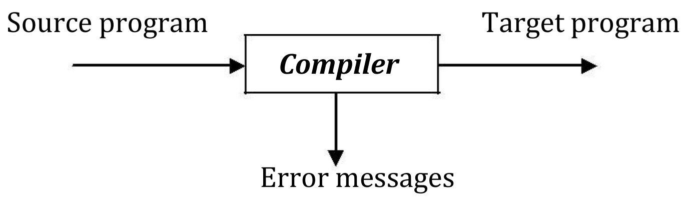
p_n { $action_n$}

- 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:

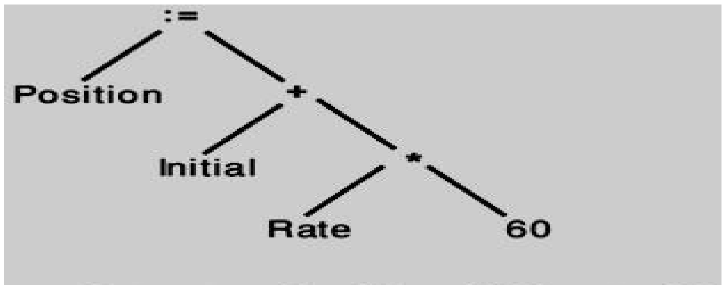
There are two parts to compilation as

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.***

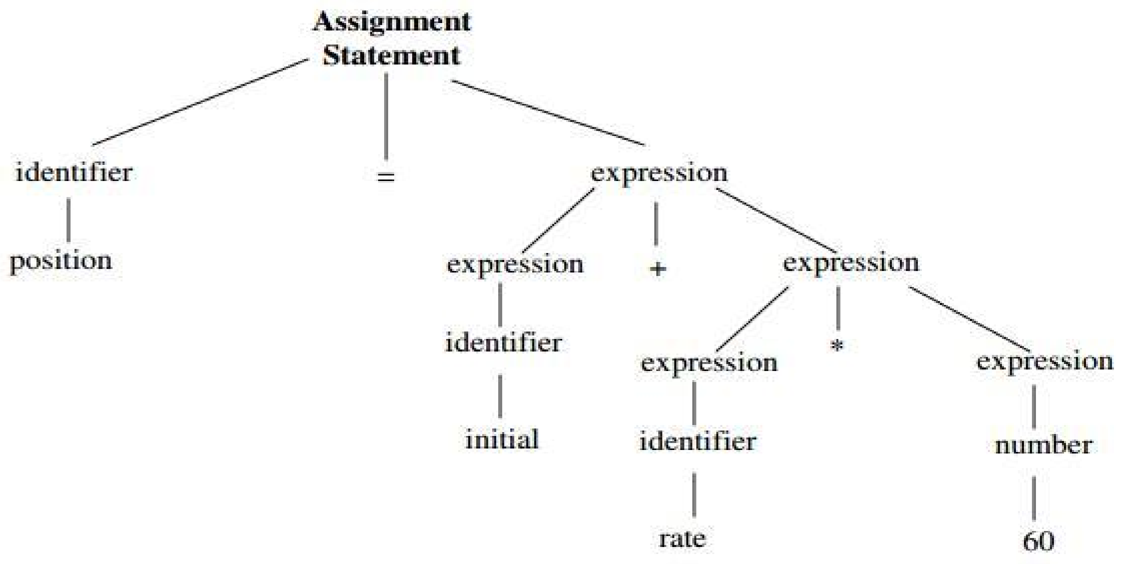
For example, in lexical analysis the characters in the assignment statement **position: = initial + rate * 60** would be grouped into the following tokens:

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 *expression*$_1$ and *expression*$_2$ are expressions, then so are

- *expression*$_1$ + *expression*$_2$
- *expression*$_1$ * *expression*$_2$
- (*expression*$_1$)

- 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 *identifier*$_1$ is an identifier and *expression*$_2$ is an expression, then

identifier$_1$:= *expression*$_2$

is a statement.

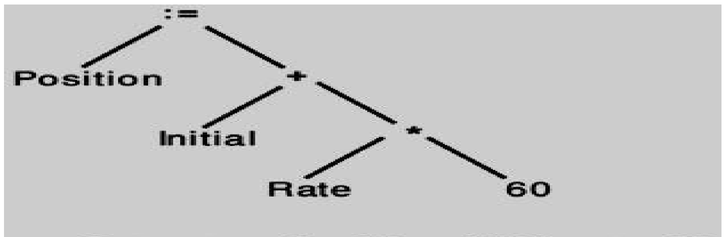
2. If *expression*$_1$ is an expression and *statement*$_2$ is a statement, then

while *(expression*$_1$*)* ***do*** *statement*$_2$

if *(expression*$_1$*)* ***then*** *statement*$_2$

are statements.

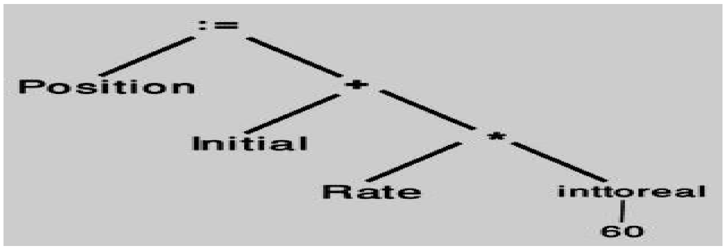
A ***syntax tree*** is a compressed representation of the parse tree in which the operators appear as the interior nodes and the operands of an operator are the children of the node for that operator.



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)

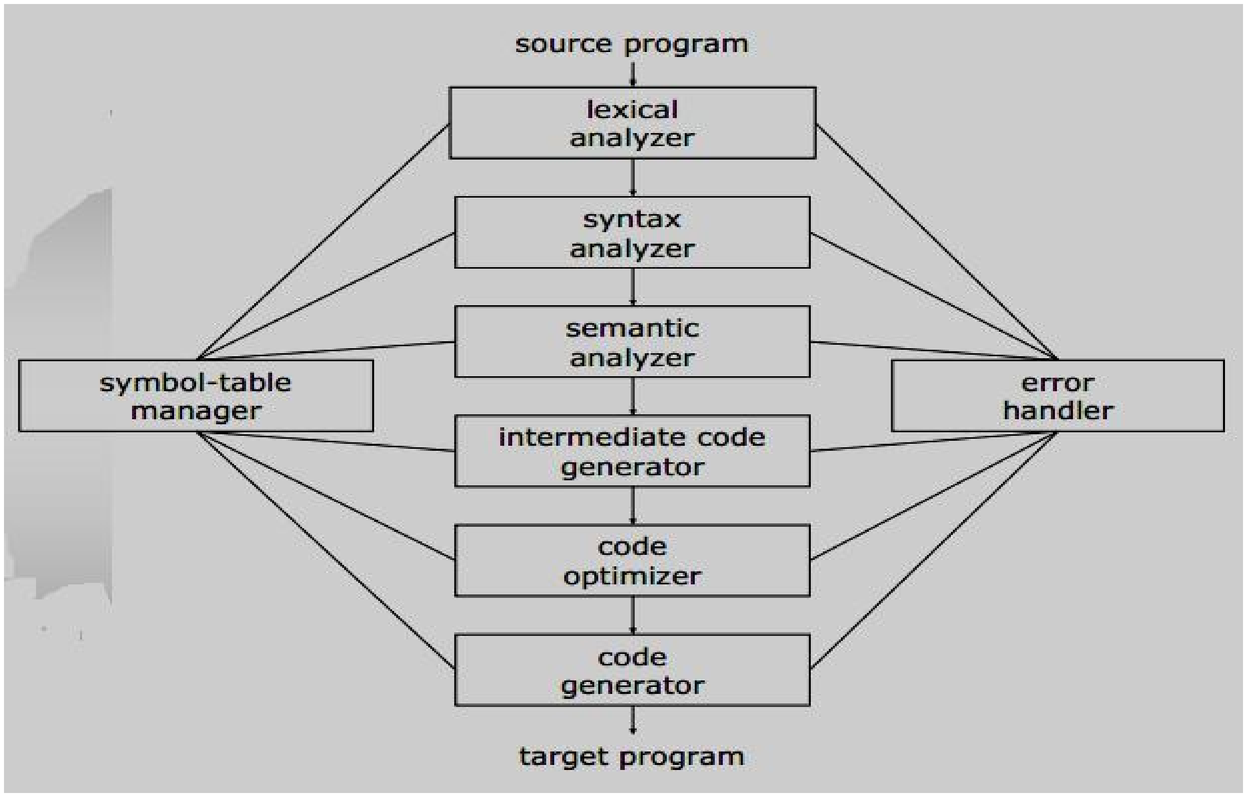
A compiler operates in phases, each of which transforms the source program from one representation to another. The six phases of a complier are

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:

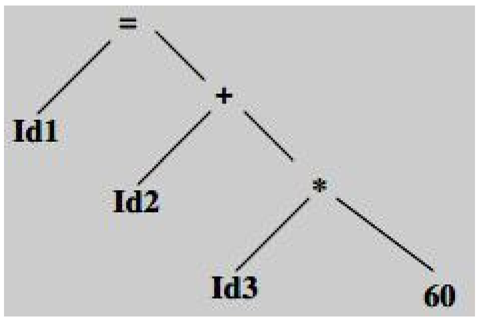
- The ***lexical analysis*** phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if, while, etc.), a punctuation character or a multi-character operator like :=.
- 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 **id_1, id_2** and **id_3** 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: **id_1: = id_2 + id_3 * 10**

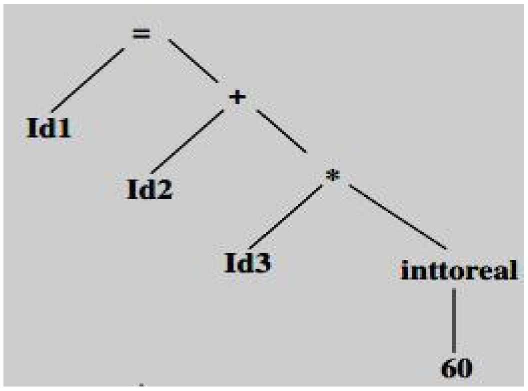
Syntax analyzer:

- ***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.

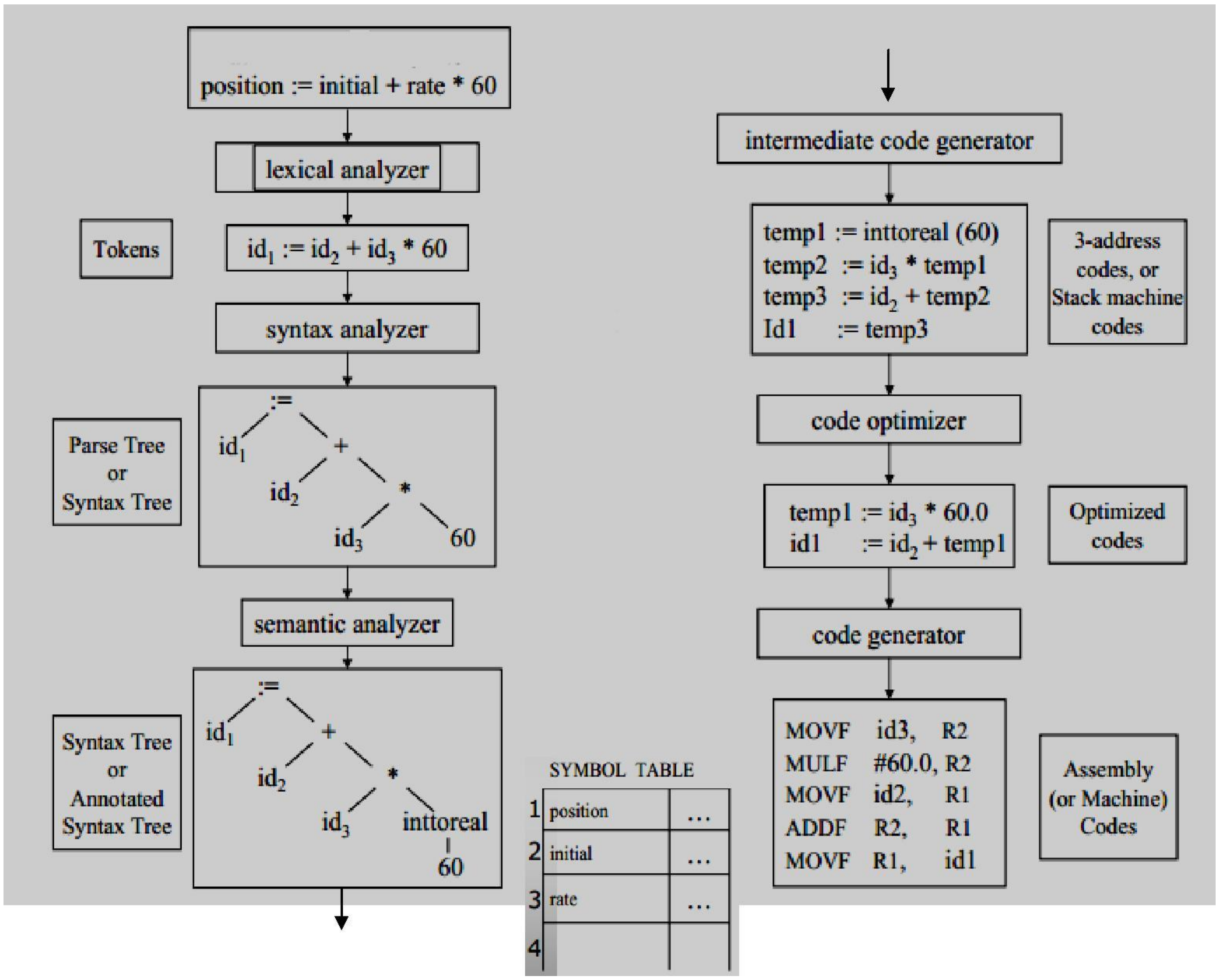
Error Detection and Reporting:

- 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.

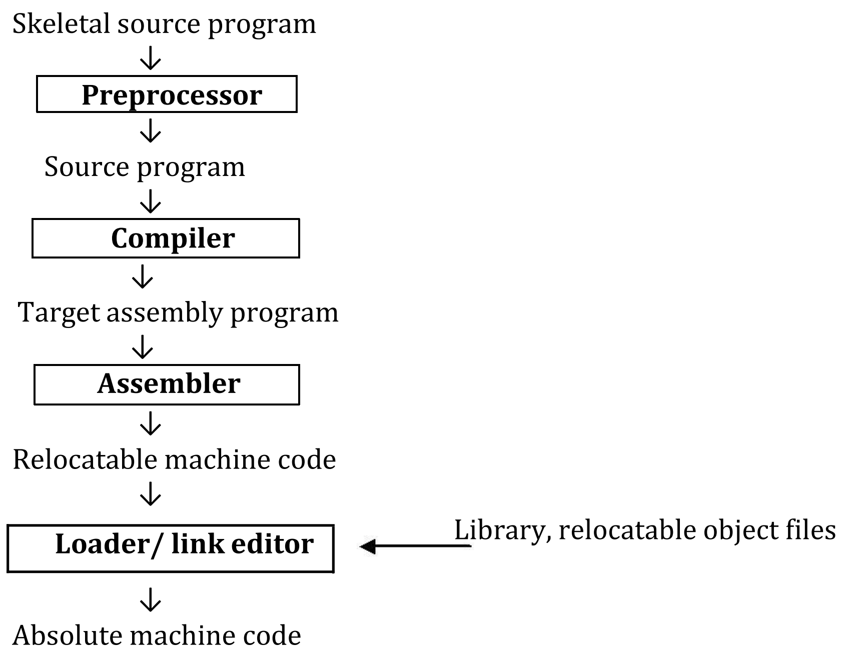
Example:

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:

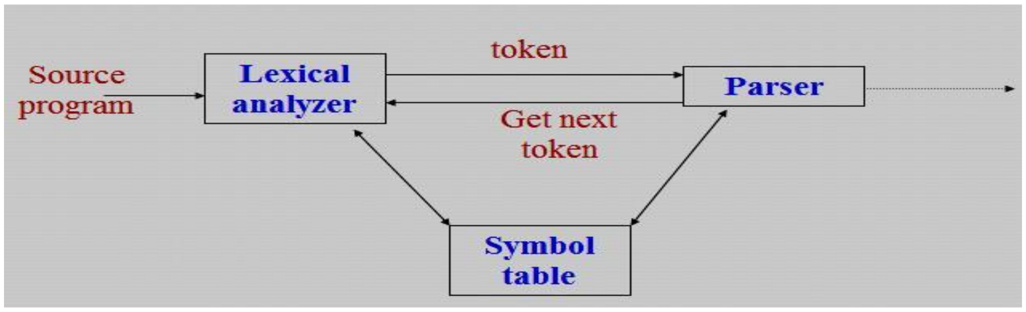
- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
- The rules must handle the different possible access methods for data.
- Eg; variables may be in registers, in a fixed location in memory 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.

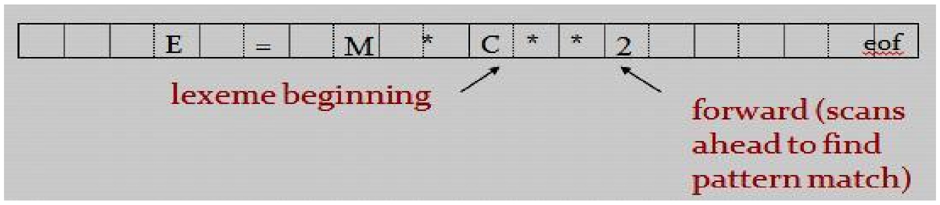
The three general approaches to the implementation of a lexical analyzer

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.

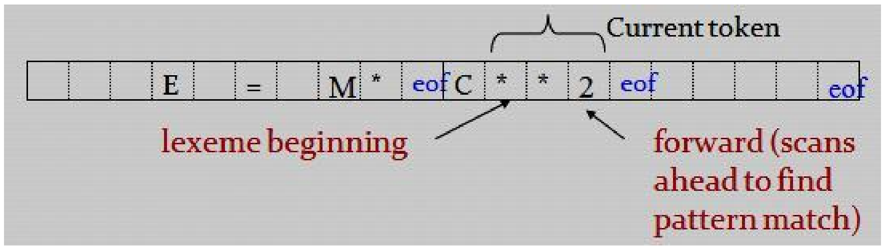
- If the forward pointer is to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

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:

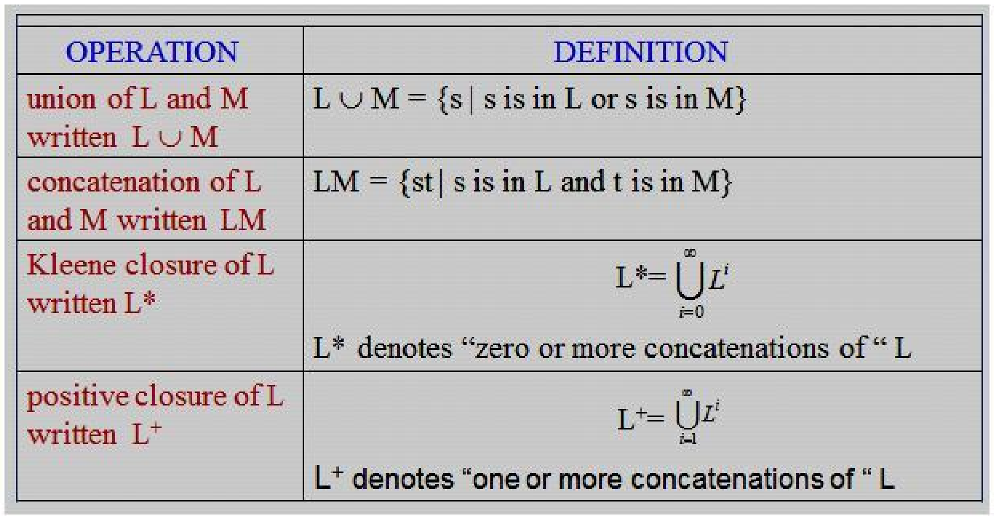
- The term *alphabet or character class* denotes any finite set of symbols. Typical examples of symbols are letters and characters.
- The set {0, 1} is the *binary alphabet.*
- ASCII and EBCDIC are two examples of computer alphabet.
- A ***string*** over some alphabet is a finite sequence of symbols 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 Φ, the empty set, or {ε},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. ε is a regular expression that denotes {ε}, the set containing empty string.
2. If ***a*** is a symbol in Σ, 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 Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

where each d_i is a distinct name, and each r_i 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.***

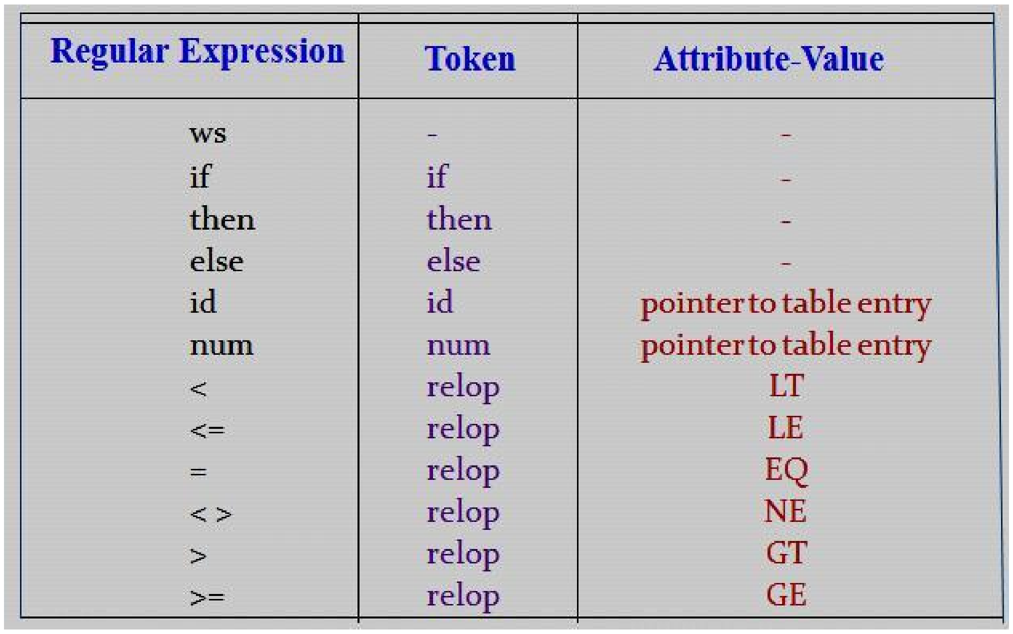
We assume lexemes are separated by **white space** consisting of *blanks, tabs, and newlines*. In lexical analyzer will strip out white space.

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

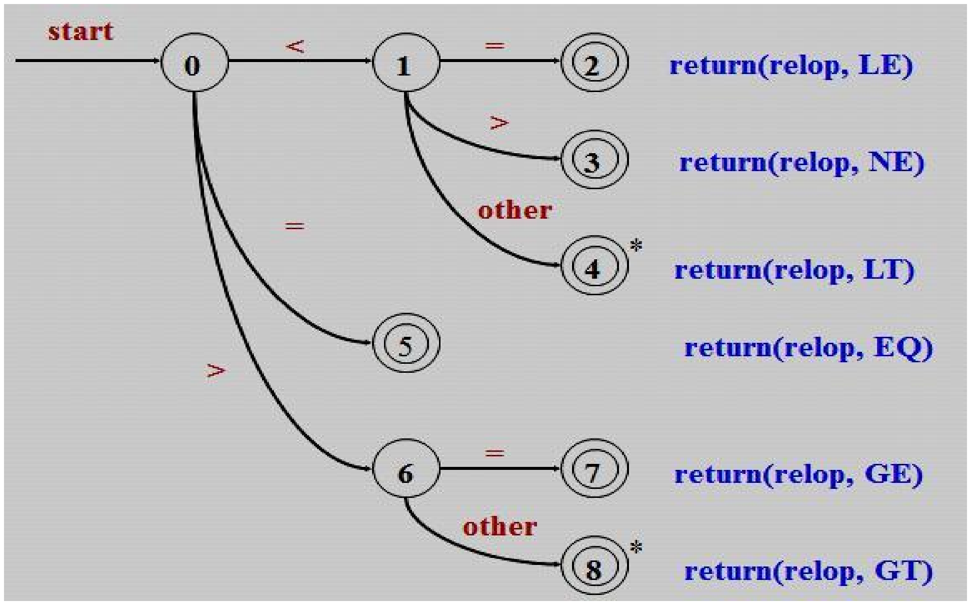
Regular expression pattern for tokens

Transition diagram:

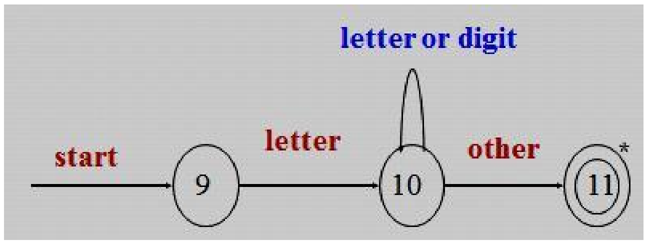
- An intermediate step in the construction of a lexical analyzer, produce a stylized flowchart called ***a transition diagram.***
- Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
- Transition diagram to keep track of information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

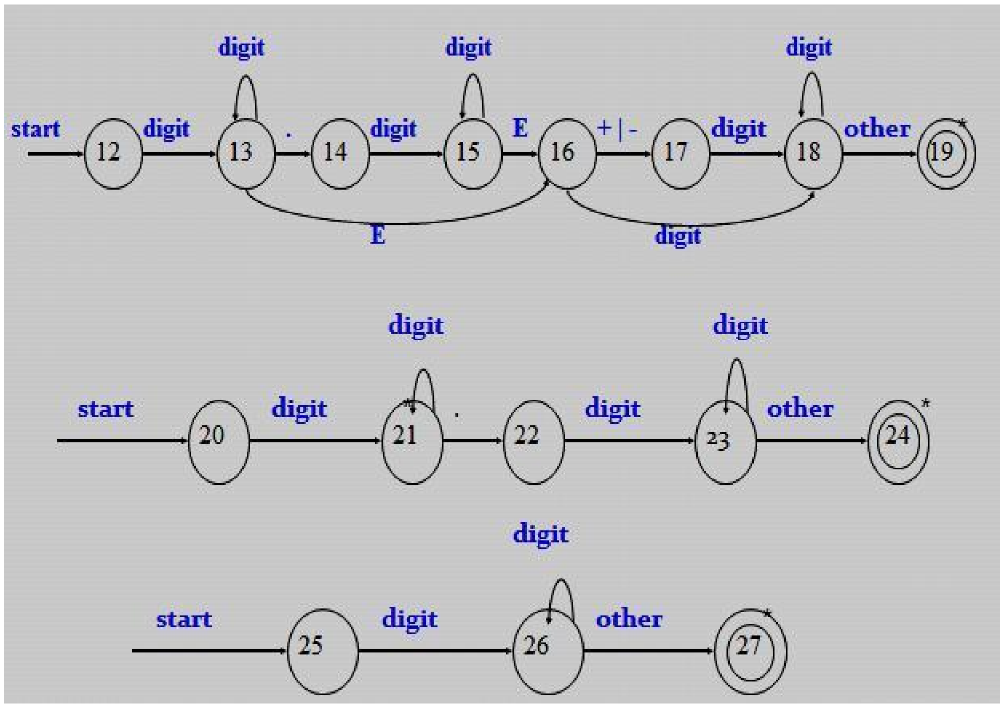
Transition diagram for relational operators:



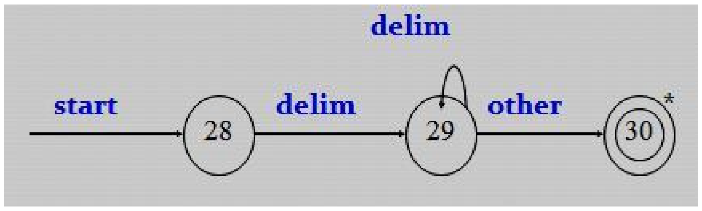
Transition diagram for identifiers and keywords:



Transition diagram for digits:



Transition diagram for delim:



Implementing a Transition diagram:

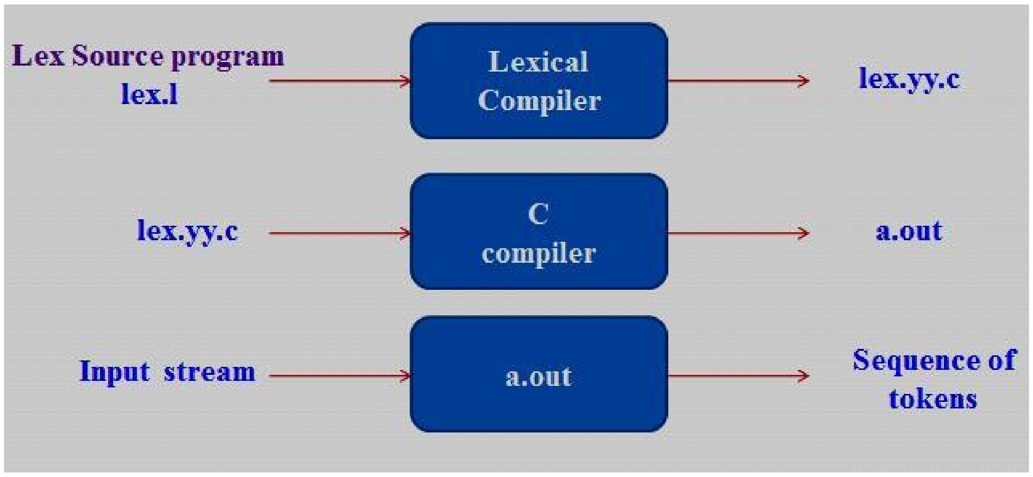
- A sequence of transition diagram can be converted into a program to look for the tokens by the diagrams.
- The systematic approach that works for all transition diagram and constructs programs whose size is proportional to the number of states and edges in the diagrams.

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 { *$action_1$*}

    p_2 { *$action_2$*}

    …… …..

    p_n { *$action_n$*}

  where each ***p_i*** is a regular expression and each ***$action_i$*** 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 ***p_i***.
- Then it executes ***$action_i$*.**
- 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

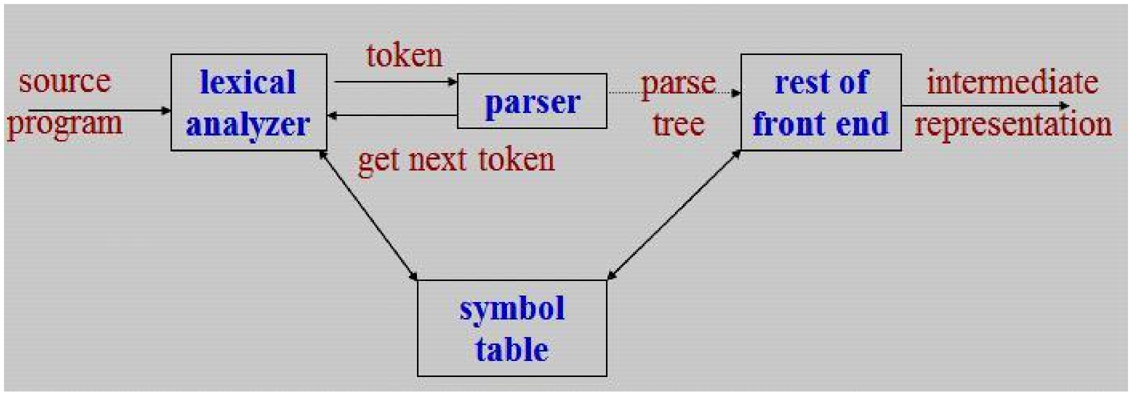
PARSING

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 $\rightarrow$ ε 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 $\rightarrow$ αBβ, then everything in FIRST (β) except for ε is placed in FOLLOW (B).
3. If there is a production A $\rightarrow$ αB, or a production A $\rightarrow$ αBβ 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

$$S = \gamma_0 \Rightarrow \gamma_1 \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

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

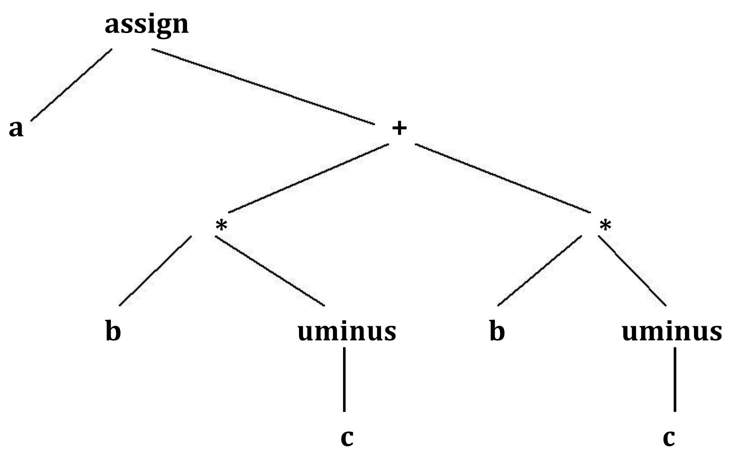
- **i.** Syntax trees
- **ii.** Postfix notation
- **iii.** Three - address code

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.

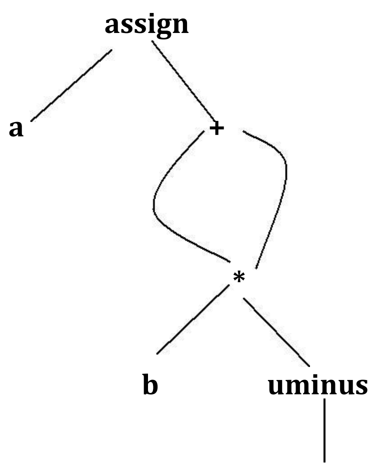
37. What is a syntax tree? Draw the syntax tree for the assignment statement.

- 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 → E ***or*** E | E ***and*** E | ***not*** E | (E) | id ***relop*** id | ***true*** | ***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:    t1 := 0
102:    goto 104
103:    t1 := 1
104:    if c< d goto 107
105:    t2 := 0
106:    goto 108
107:    t2 := 1
108:    if e< f goto 111
109:    t3 := 0
110:    goto 112
111:    t3 := 1
112:    t4 := t2 and t3
113:    t5 := t1 or t4
```

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)

$S \rightarrow 0|1|AS0|BS0 \qquad A \rightarrow \varepsilon \; B \rightarrow \varepsilon$

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}**

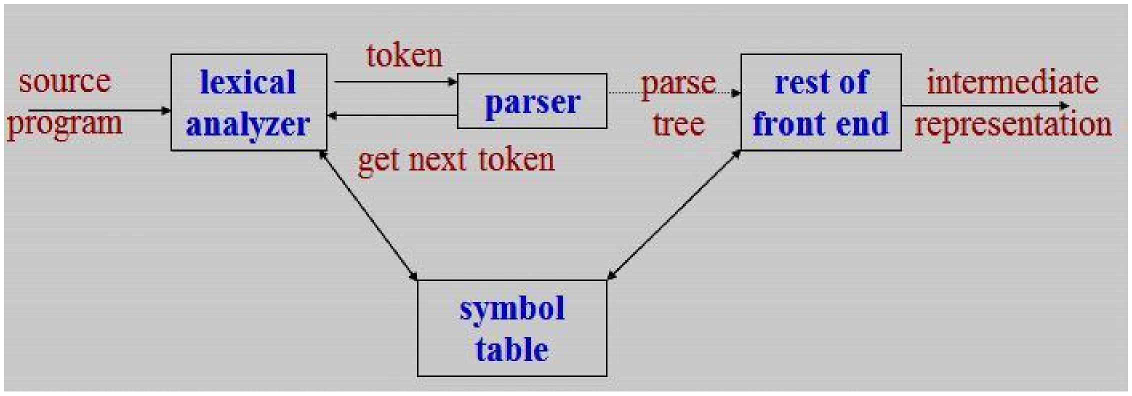
FOLLOW (A) = FOLLOW(S) = **{$, 0}**

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 $\rightarrow$ too inefficient to use in production compilers.
   - Cocke-Younger –Kasami algorithm and
   - Earley's algorithm
2. Top down parser $\rightarrow$ it builds parse trees from the top (root) to the bottom (leaves).
3. Bottom up parser $\rightarrow$ 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.

  E → E + E | E – E | E * E | (E) | - E | id

- The production E → - 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.

  E ⇒ -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*.

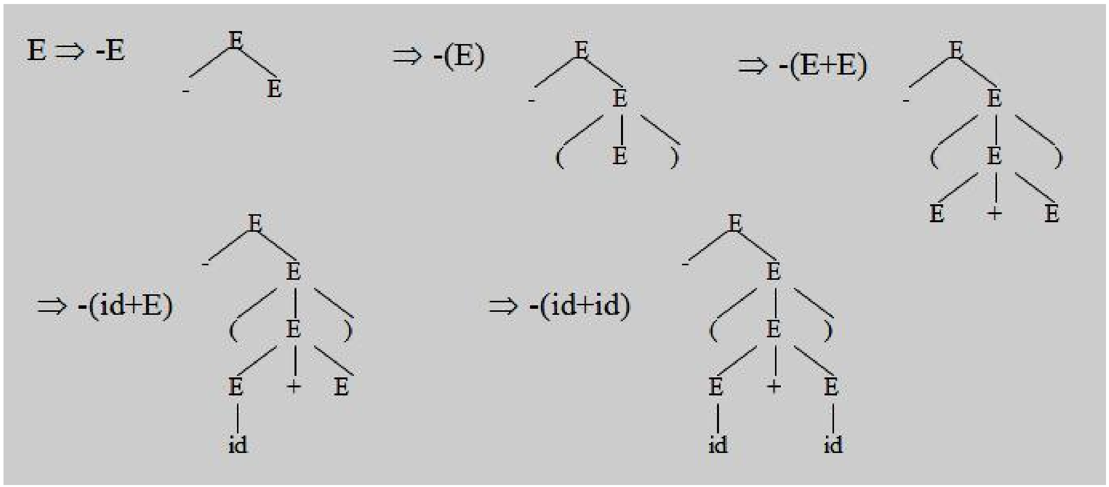
The string – (id+id) is a sentence of the grammar, and then the derivation is

E ⇒ -E ⇒ - (E) ⇒ - (E+E) ⇒ - (**id**+E) ⇒ - (**id**+**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.***



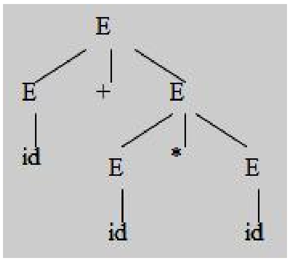
Ambiguity

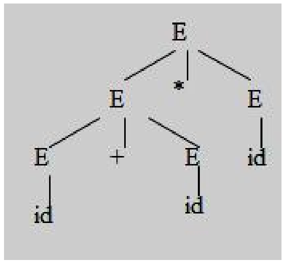
- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous***.
- An ***ambiguous grammar*** is one that produces more than one left most or more than one right most derivation for the same sentence.
- For the most parsers, the grammar must be unambiguous.
- The ***unambiguous grammar*** unique selection of the parse tree for a sentence.

The sentence **id+id*id** has the two distinct *leftmost derivations:*

$E \Rightarrow E + E$	$E \Rightarrow E * E$
$\Rightarrow id + E$	$\Rightarrow E + E * E$
$\Rightarrow id + E * E$	$\Rightarrow id + E * E$
$\Rightarrow id + id * E$	$\Rightarrow id + id * E$
$\Rightarrow id + id * id$	$\Rightarrow id + id * id$

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

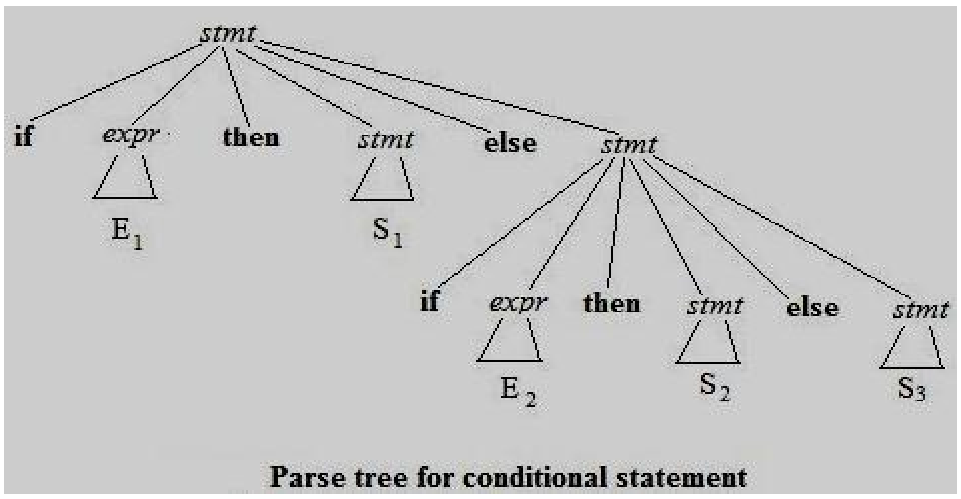
  | **if** expr **then** stmt **else** stmt

  | **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

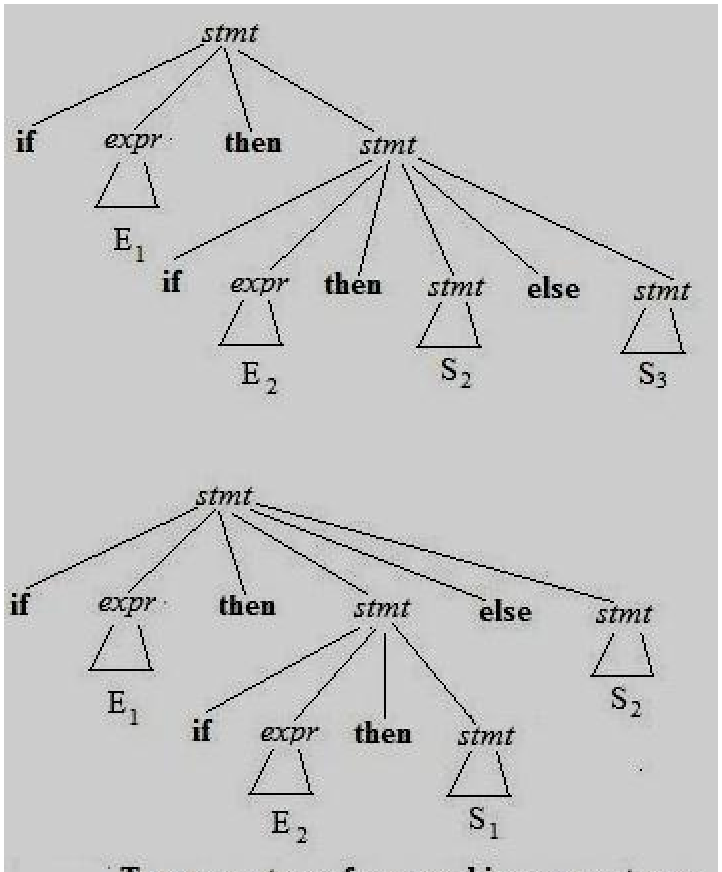


Parse tree for conditional statement

Grammar is ambiguous since the string

if *E1* ***then if*** *E2* ***then*** *S1* ***else*** *S2* 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 $A_1, A_2, .... A_n$
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 $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$
              where $A_j \rightarrow \delta_1 \mid \delta_2 \mid ... \mid \delta_k$ 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 $\mathbf{A} \rightarrow \boldsymbol{\alpha\beta_1} \mid \boldsymbol{\alpha\beta_2}$ then it is left factored as:

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

$\mathbf{A}' \rightarrow \boldsymbol{\beta_1} \mid \boldsymbol{\beta_2}$

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:

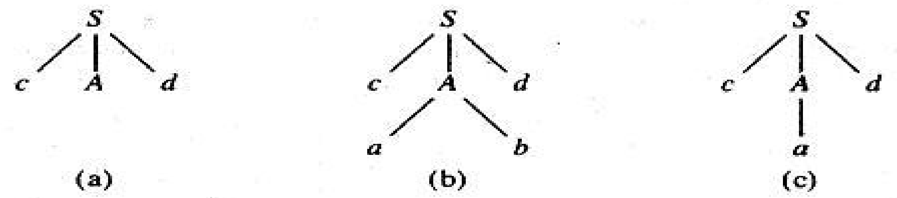
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- It is to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- The special case of recursive-descent parsing called predictive parsing, where 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 $\rightarrow$ **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 $\rightarrow$ 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.

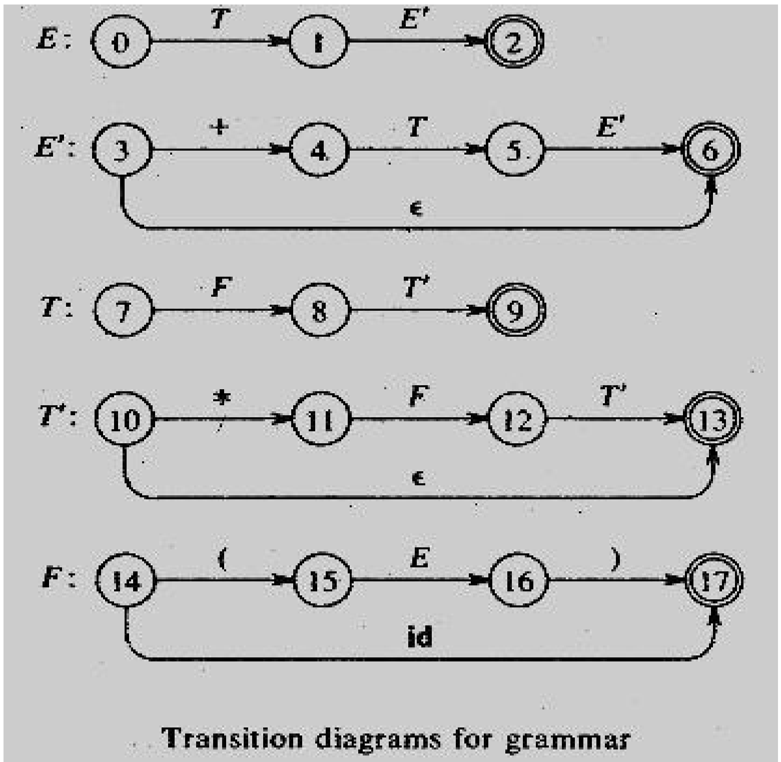
- Transition diagrams can be simplified by substituting diagrams in one another; these substitutions are similar to the transformations on grammars.

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Consider the following grammar for arithmetic expressions,

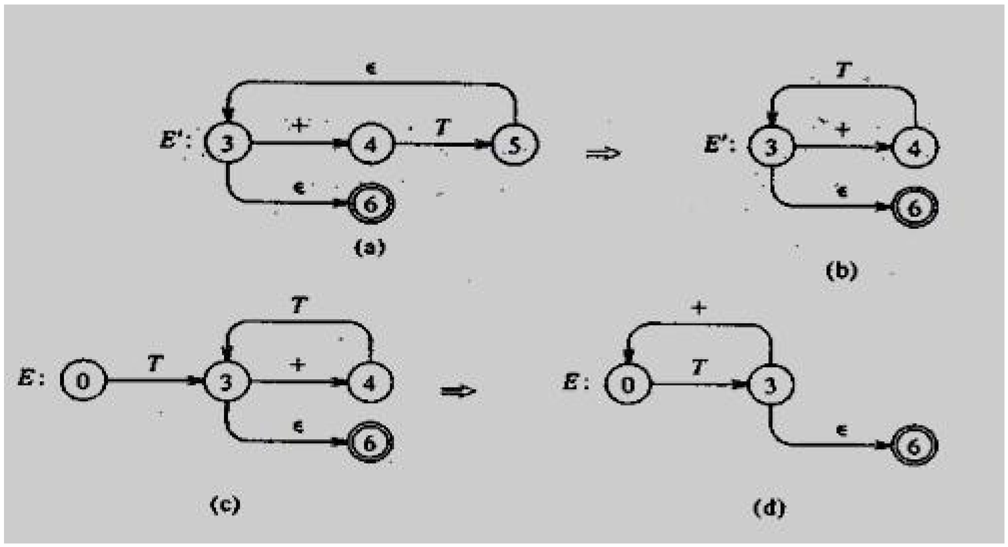
$$E \rightarrow T$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

Transition diagrams for grammar:

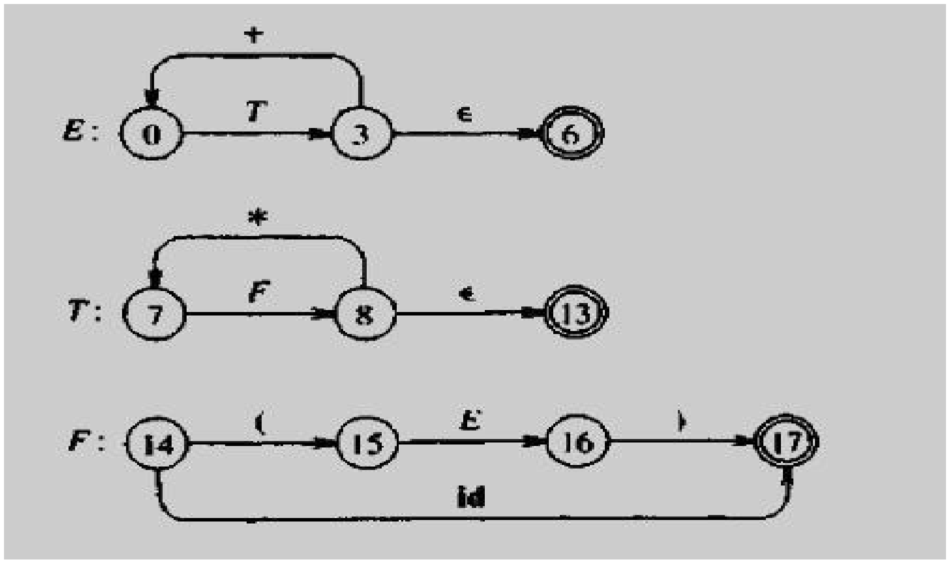


Transition diagrams for grammar

Simplified transition diagram:

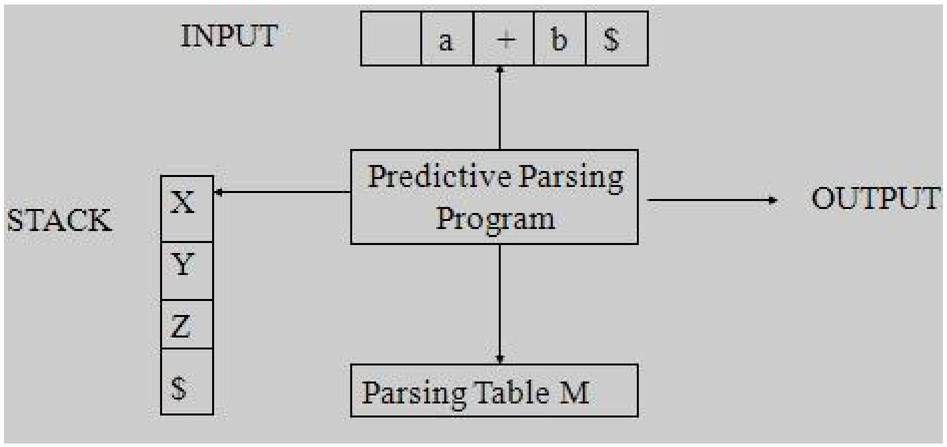


Simplified transition diagrams for arithmetic expressions:



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 $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Compute FIRST and FOLLOW and also find the parsing table. The input string is id+id * id.

<u>Solution:</u>

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** }

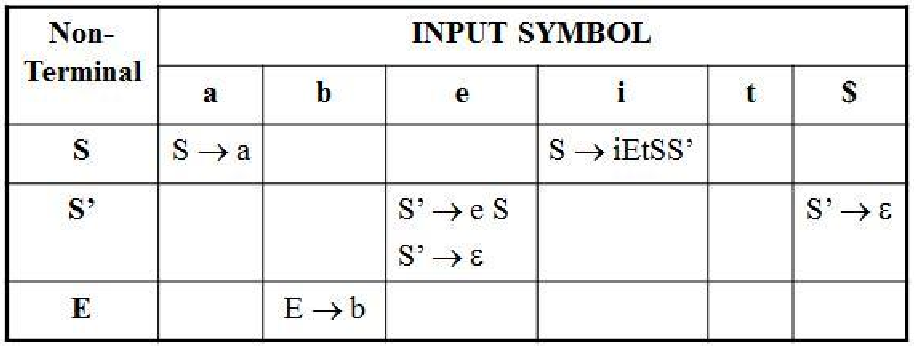
3. COMPUTATION OF FOLLOW:

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 → 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***.

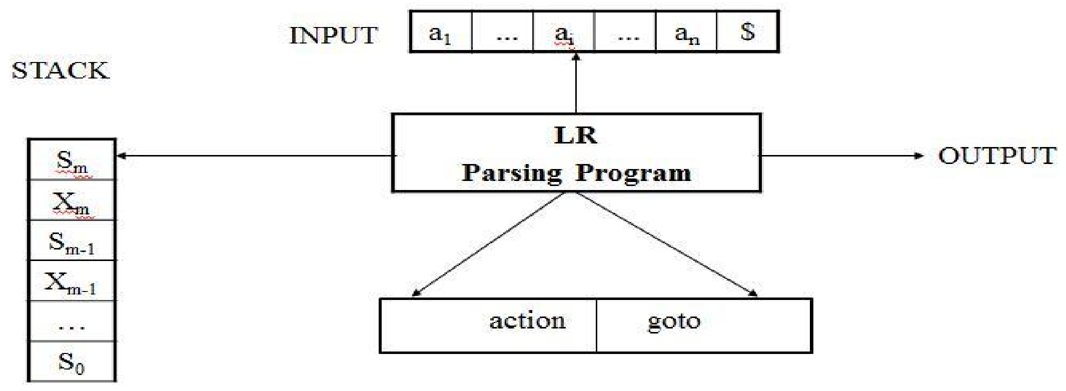
Three techniques are used for constructing an LR parsing table for a grammar.

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 → 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 → 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(I6, ( )

I4: F -> (.E)
E -> .E + T
E -> .T
T -> . T * F
T -> .F
F -> .(E)
F -> . id

goto(I6, id)

I5: F -> id.

goto(I7, F)

I10: T -> T * F.

goto(I7, ( )

I4: F -> (.E)
E -> .E + T
E -> .T
T -> .T * F
T -> .F
F -> .(E)
F -> .id

goto(I7, id)

I5: F -> id.

goto(I8,))

I11: F -> (E).

goto(I8, +)

I6: 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** {+} = { **+,), $** }

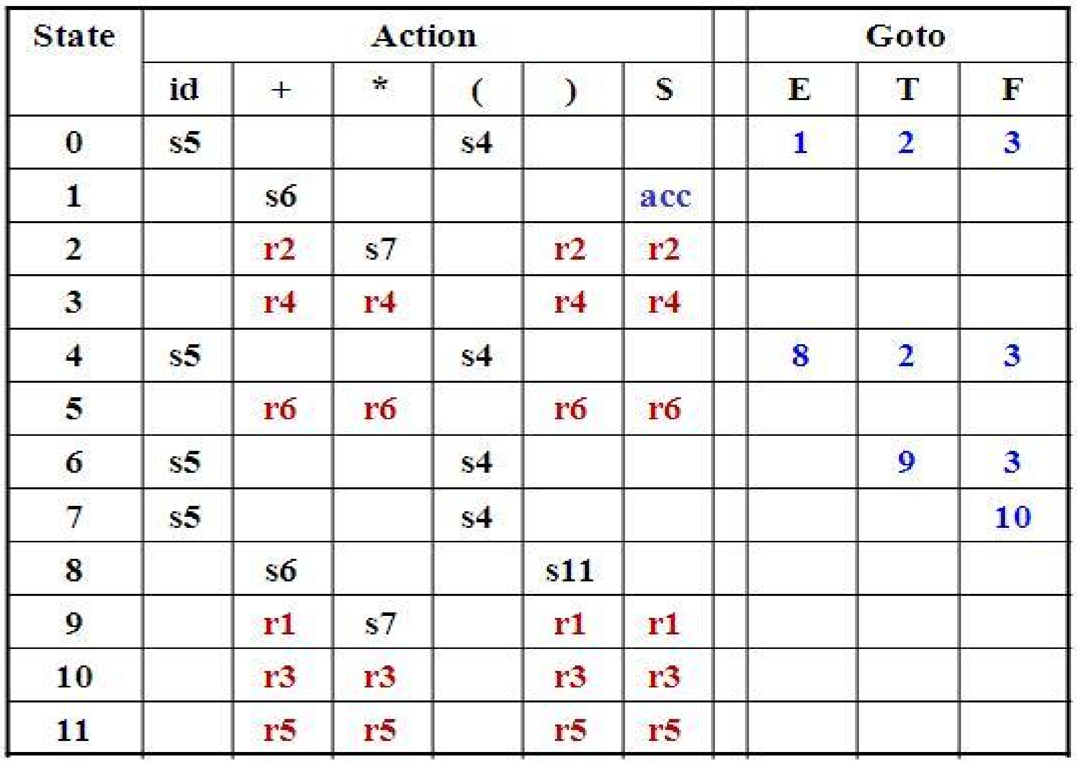
FOLLOW (T') = FOLLOW (T) = { **+,), $** }

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	+	*	(	)	$	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 $**	action [5, +] = r6 then **reduce r6:** **F -> id** 1) POP **2 symbols** from the stack 2) goto [0, F] = **3** 3) Push **'F3'** into the stack
(3)	**OF3**	**+id$**	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)

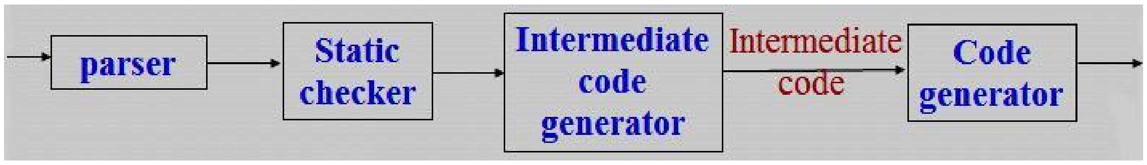
In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.

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

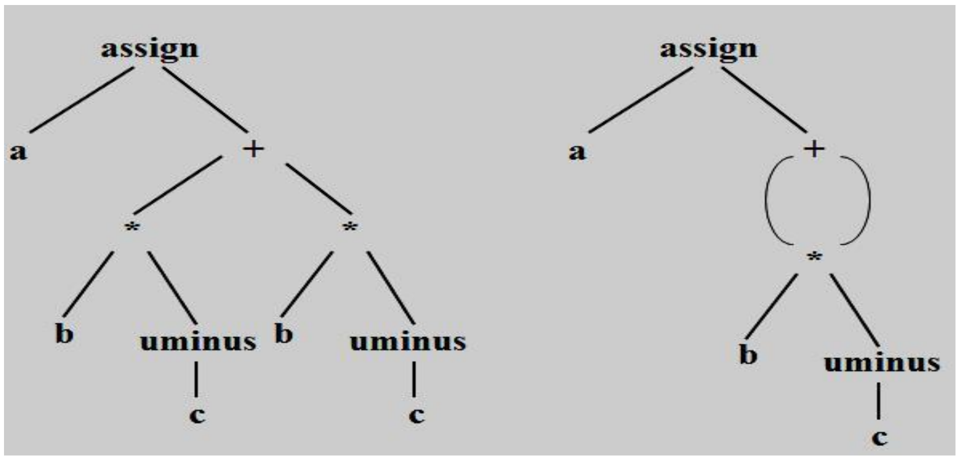
i. ***Syntax trees***
ii. ***Postfix notation***
iii. ***Three - address code***

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)***

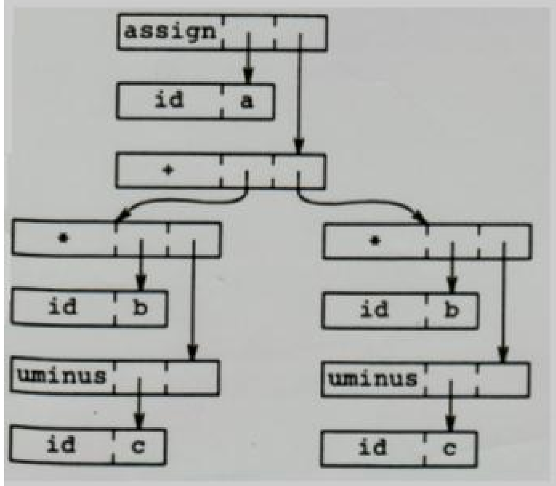
- return a pointer to an existing node whenever possible, instead of constructing new nodes.
- The token ***id*** has an attribute place that points to the symbol-table entry for the identifier.
- The symbol table entry can be found from an attribute ***id.name***, representing the lexeme associated with that occurrence of ***id***.

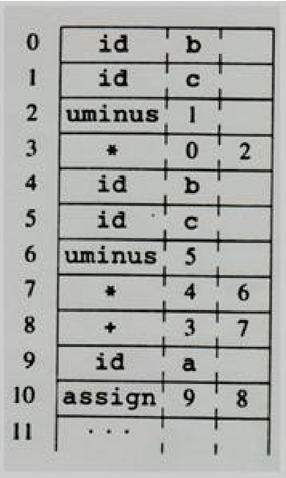
PRODUCTION	**SEMANTIC RULES**
S → id: = E	S.nptr := mknode ('assign', mkleaf (id, id.entry), E.nptr)
E → E_1 + E2	E.nptr := mknode('+', E_1.nptr,E_2.nptr)
E → E_1 * E2	E.nptr := mknode('*', E_1.nptr,E_2.nptr)
E → - E_1	E.nptr := mknode('uminus',E_1.nptr)
E → (E1)	E.nptr := E_1.nptr
E → id	E.nptr := mkleaf(id, 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.





0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	. . .		

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.

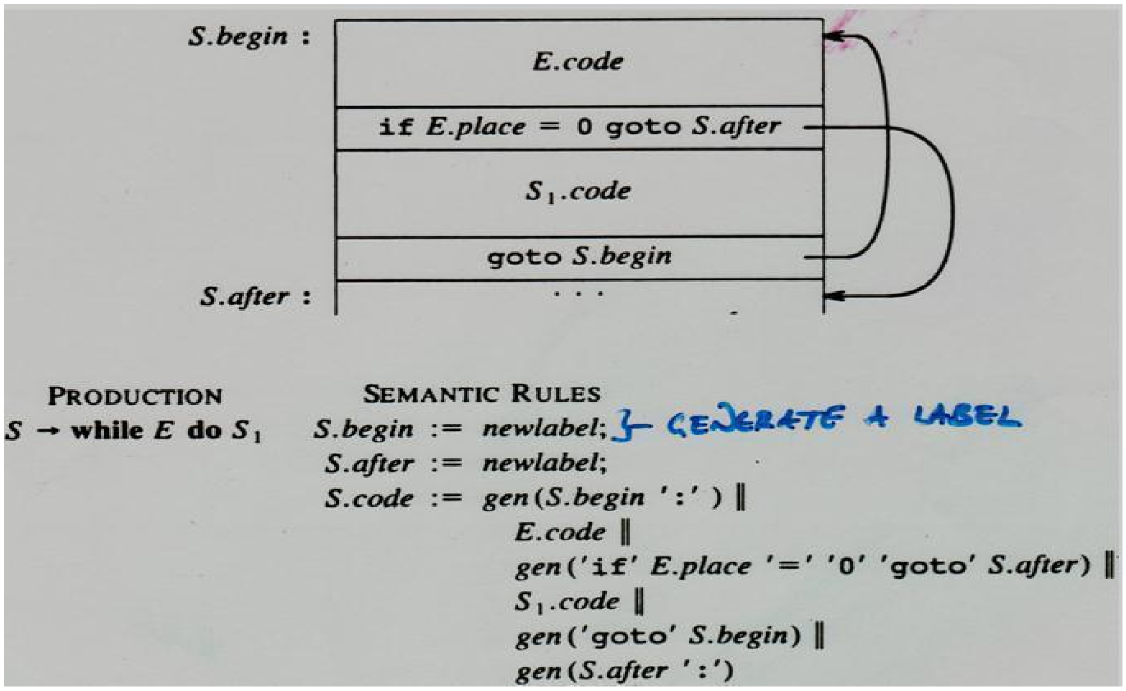
12. Explain the syntax directed translation into three- address code? (5 marks)

- 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*** $\rightarrow$ ***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*** $\rightarrow$ ***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 → id := E	S.code := E.code || **gen**(**id.**place ':=' E.place)
E → 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 → 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 → - E_1	E.place:= newtemp ; E.code := E_1.code || **gen**(E.place ':=' **'uminus'** E_1.place)
E → (E_1)	E.place:= E_1.place ; E.code := E_1.code
E → 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



PRODUCTION	SEMANTIC RULES
S → **while** E **do** S_1	S.begin := newlabel; } GENERATE A LABEL
	S.after := newlabel;
	S.code := gen(S.begin ':') ||
	E.code ||
	gen('if' E.place '=' '0' 'goto' S.after) ||
	S_1.code ||
	gen('goto' S.begin) ||
	gen(S.after ':')

Semantic rules generating code for a while statement

13. Explain the three implementation of three-address statements? (11 marks)

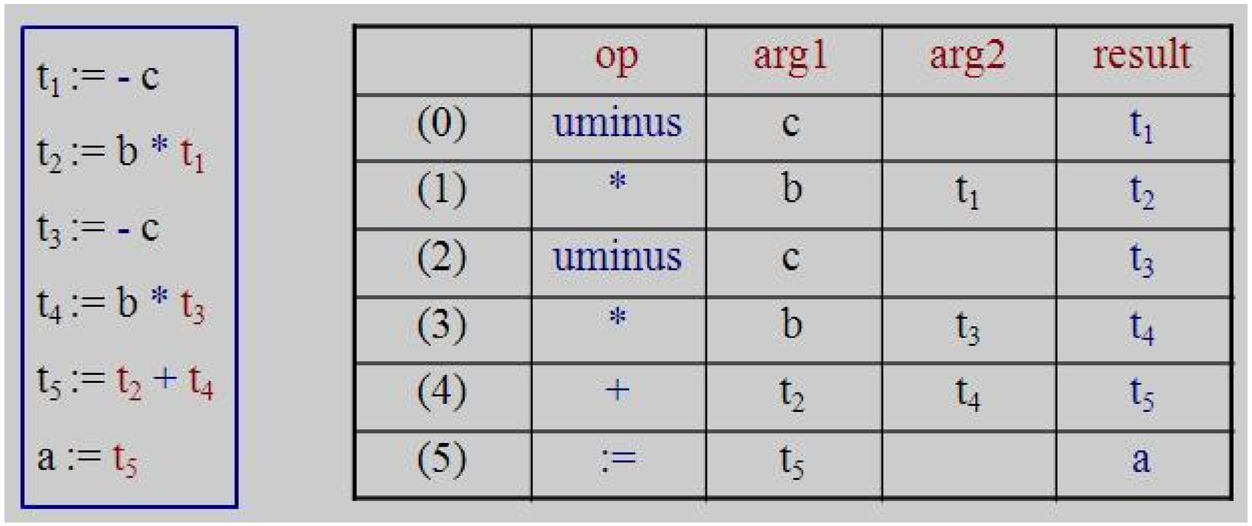
Implementations of three-Address Statements

- 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***



$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

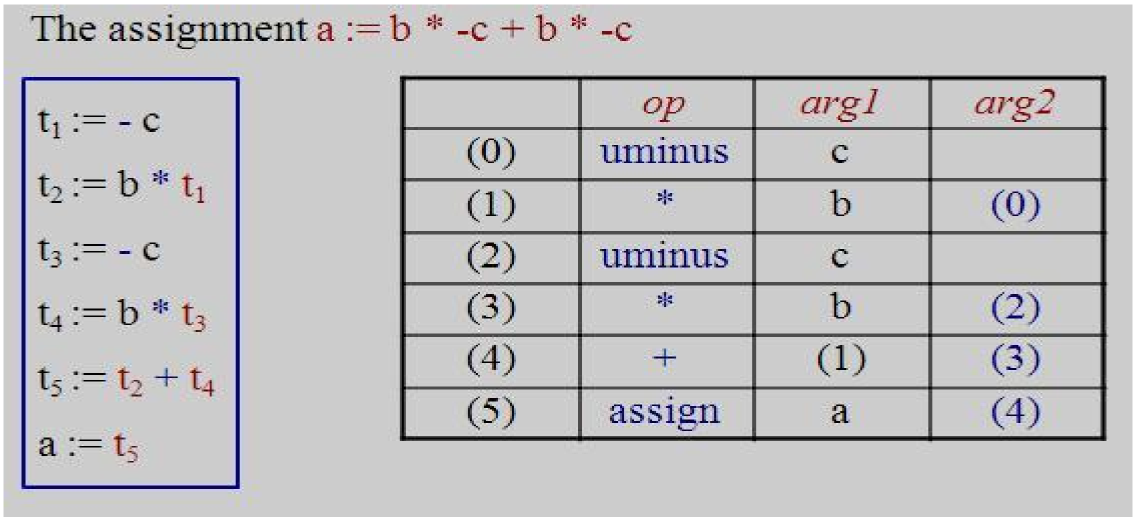
	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

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.



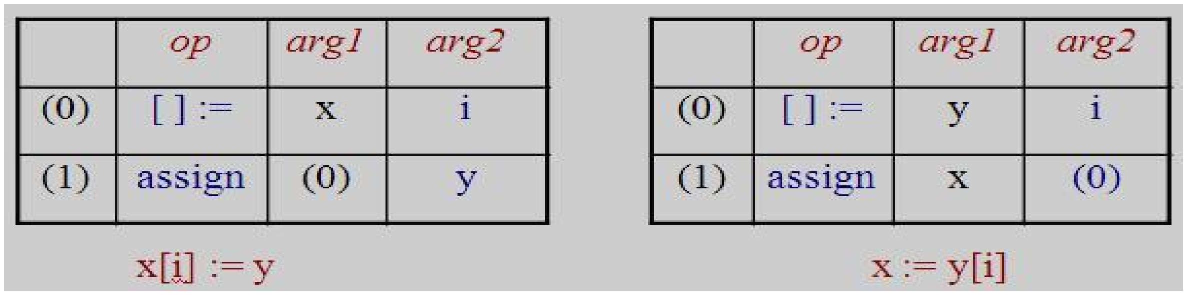
The assignment a := b * -c + b * -c

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

	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: = t₅*** 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

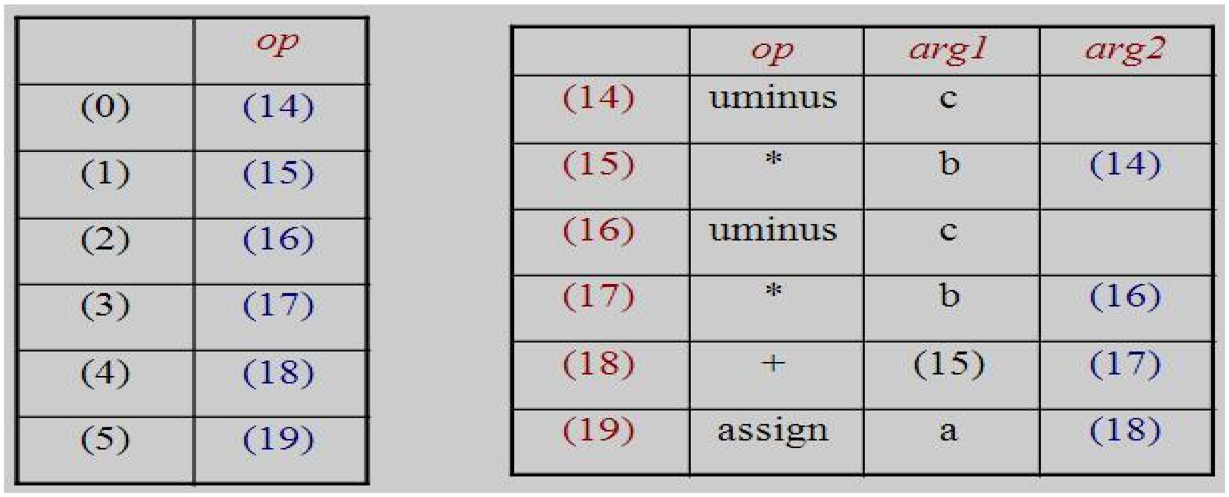
x[i] := y

	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)

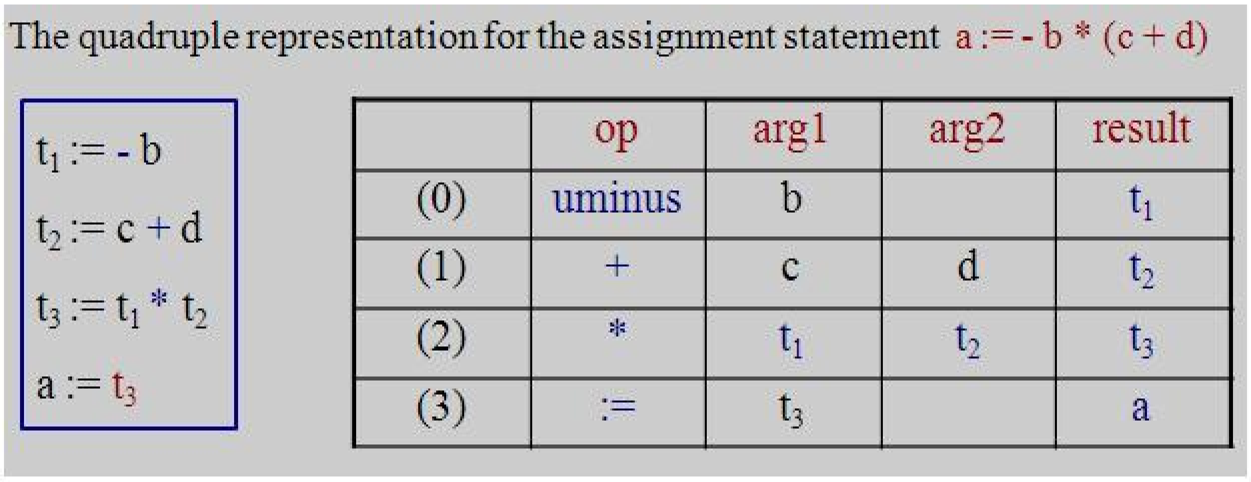
	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

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)

$t_1 := - b$

$t_2 := c + d$

$t_3 := t_1 * t_2$

$a := t_3$

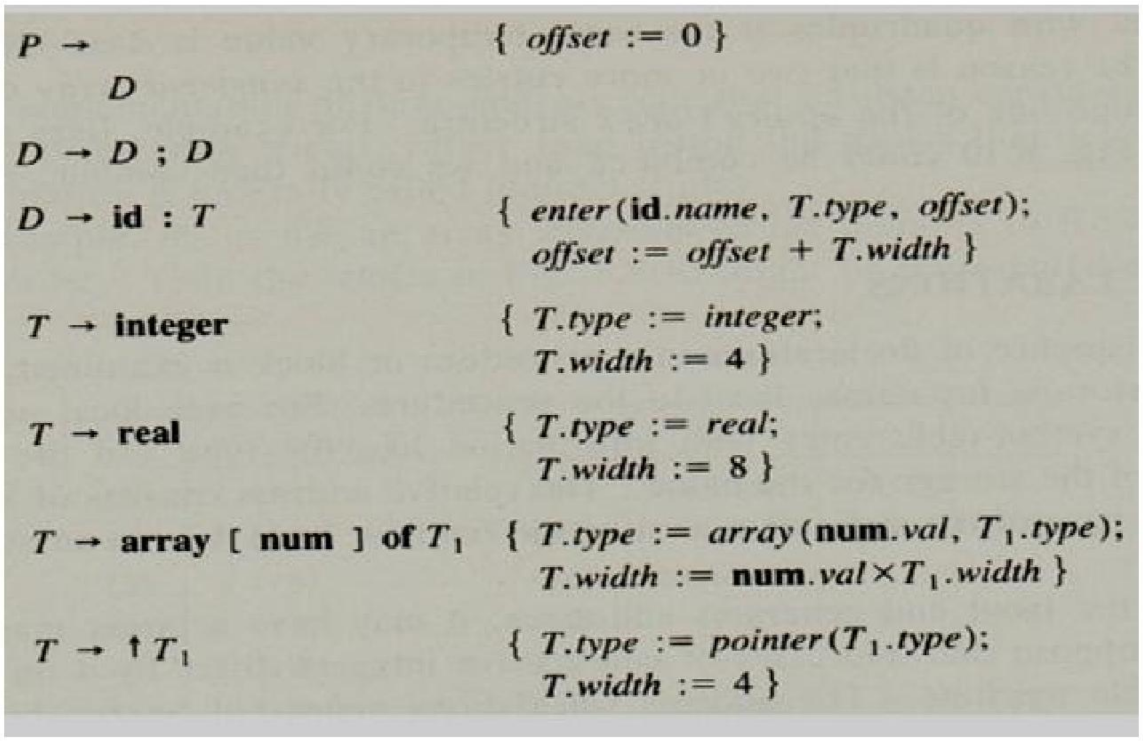
	op	arg1	arg2	result
(0)	uminus	b		t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3		a

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.***



$P \rightarrow$	$\{ \ offset := 0 \ \}$
$\quad D$	
$D \rightarrow D \ ; \ D$	
$D \rightarrow \textbf{id} : T$	$\{ \ enter(\textbf{id}.name,\ T.type,\ offset);$ $offset := offset + T.width \ \}$
$T \rightarrow \textbf{integer}$	$\{ \ T.type := integer;$ $T.width := 4 \ \}$
$T \rightarrow \textbf{real}$	$\{ \ T.type := real;$ $T.width := 8 \ \}$
$T \rightarrow \textbf{array} \ [\ \textbf{num} \ ] \ \textbf{of} \ T_1$	$\{ \ T.type := array(\textbf{num}.val,\ T_1.type);$ $T.width := \textbf{num}.val \times T_1.width \ \}$
$T \rightarrow \uparrow T_1$	$\{ \ T.type := pointer(T_1.type);$ $T.width := 4 \ \}$

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

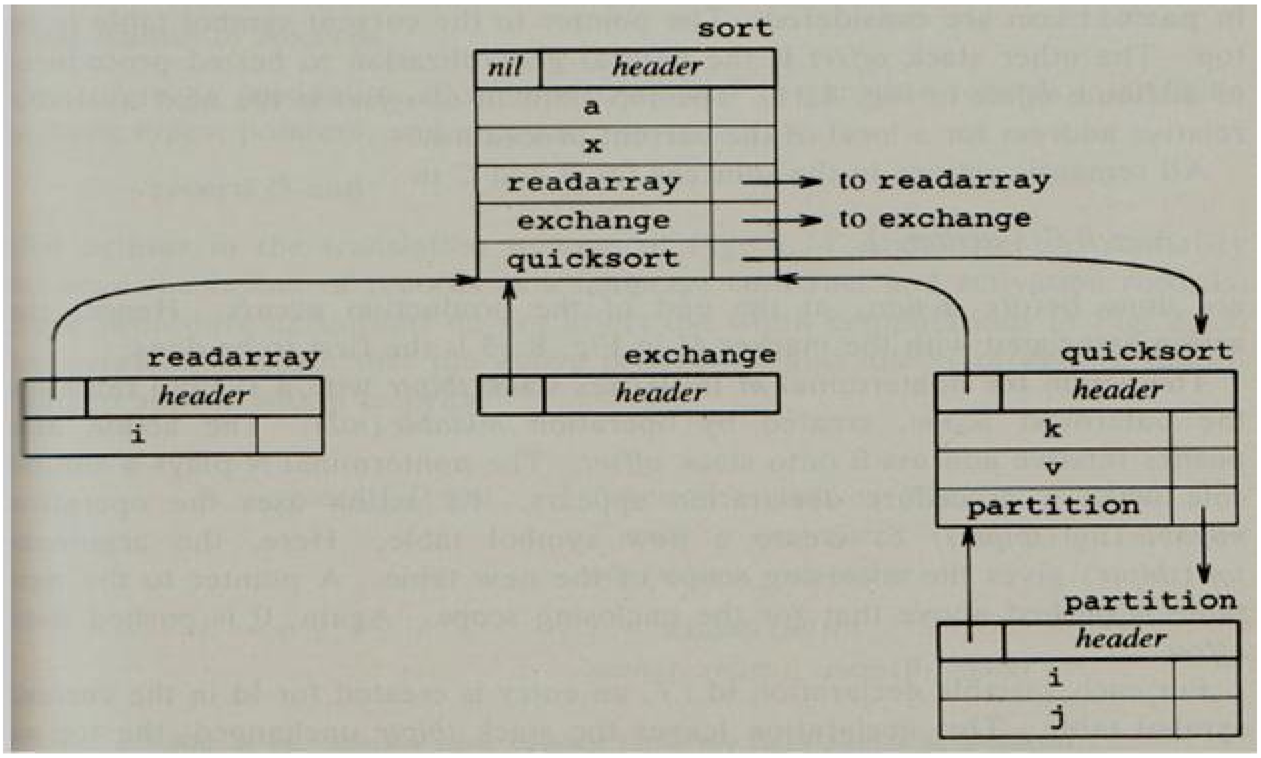
- The production for non-terminals S for statements and T for types. The non-terminal T has synthesized attributes type and width. For simplicity, suppose that there is a separate symbol table for each procedure in the language.

- A new symbol table is created when a procedure declarations **D** →**proc id D**$_1$; **S** and entries for the declarations in D$_1$ 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 → 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
else 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**

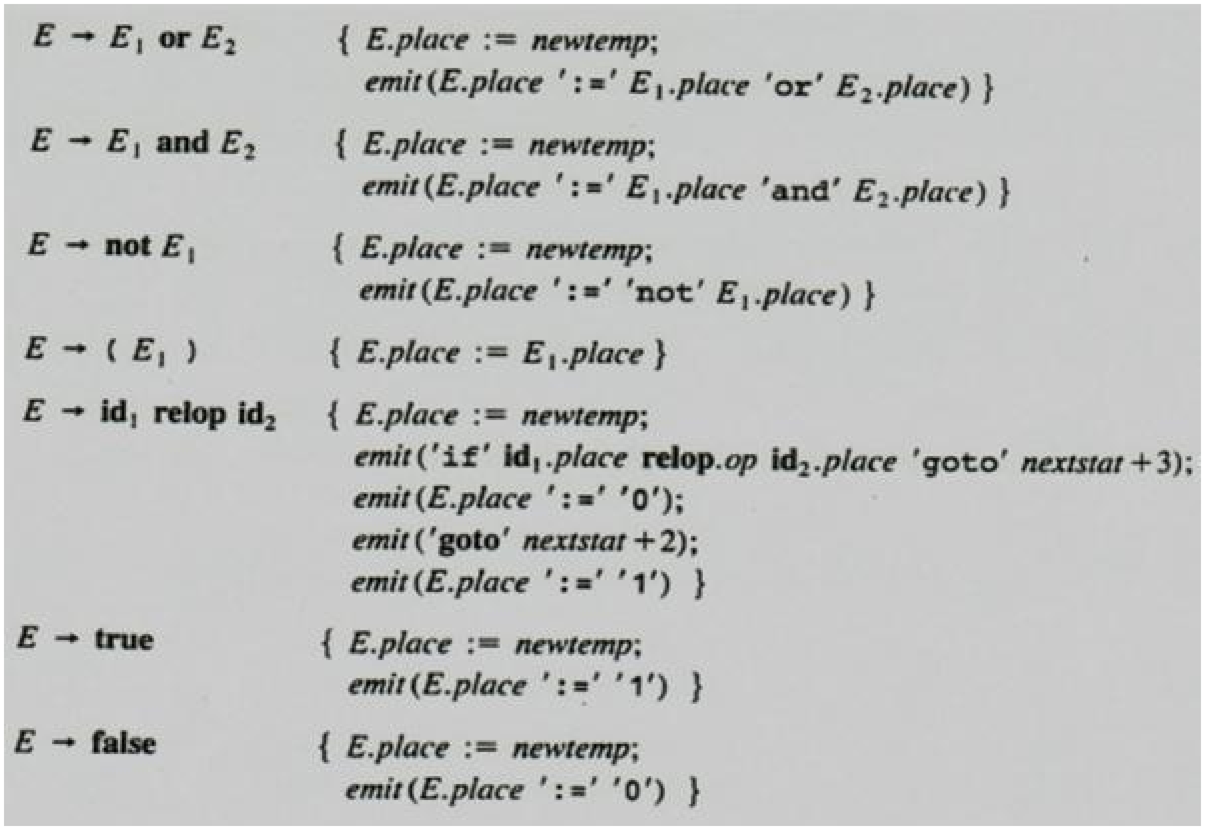
The translation for 3 address sequence is

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

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

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

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

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

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

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

- 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:    t1 := 0
102:    goto 104
103:    t1 := 1
104:    if c< d goto 107
105:    t2 := 0
106:    goto 108
107:    t2 := 1
108:    if e< f goto 111
109:    t3 := 0
110:    goto 112
111:    t3 := 1
112:    t4 := t2 and t3
113:    t5 := t1 or t4
```

(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

$$\begin{aligned} S &\rightarrow \textbf{if } E \textbf{ then } S_1 \\ &\mid \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \\ &\mid \textbf{while } E \textbf{ do } S_1 \end{aligned}$$

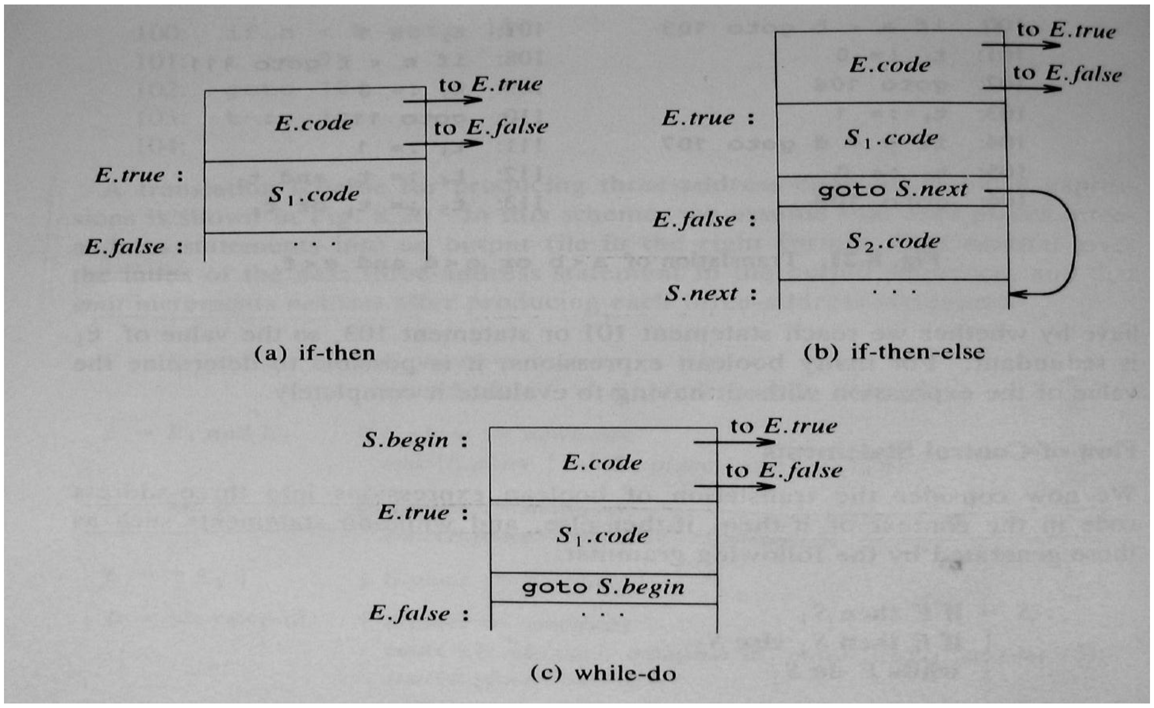
In the translation, we assume that a three-address code statement can have a symbolic label, and that the function newlabel generates such labels.

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 $\rightarrow$ 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 E1.type := arith and E2.type := arith then begin
    E.place := newtemp;
    E.code := E1.code || E2.code ||
            gen(E.place' :=' E1.place ' +' E2.place)
end

esle if E1.type := arith and E2.type := bool then begin
    E.place:= newtemp;
    E2.true := newlabel;
    E2.false := newlabel;
    E.code := E1.code || E2.code ||
            gen(E2.true' :' E.place ' :=' E1.place + 1)
            || gen('goto' nextstat + 1) ||
            gen(E2.false' :' E.place ' :=' E1.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 **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 **V_i** of the case constant.
- Each statement ***case V_i : S_i,*** creates label **L_i,** followed by code for **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
L1:           code for S1
              goto next
L2:           code for S2
              goto next
              ..........
L n-1:        code for Sn-1
              goto next
Ln:           code for Sn
              goto next
test:         if t = V1 goto L1
              if t = V2 goto L2
              .........
              if t = Vn-1 goto Ln-1
              goto Ln
next:
```

Another Translation of case statement

```
              Code to evaluate E into t
              if t ≠ V1 goto L1
              code for S1
              goto next
L1:           if t ≠ V2 goto L2
              code for S2
              goto next
L2:           ...
Ln-2:         if t ≠ Vn-1 goto Ln-1
              code for Sn-1
              goto next
Ln:           code for Sn
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 $\rightarrow$ 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 $\rightarrow$ return address is placed after code sequence itself.
- Parameters passed by reference.
- 3 address code $\rightarrow$ 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 → E₁ and M E₂**.

The semantic actions as

PRODUCTION	SEMANTIC RULES

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

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

E → not E_1	E.truelist = E_1.falselist
	E.falselist = E_1.truelist

E → (E_1)	E.truelist = E_1.truelist
	E.falselist = E_1.falselist

E → id_1 relop id_2	E.truelist = makelist(nextquad)E.falselist =
	makelist(nextquad +1)
	emit(if **id_1**.place **relop.op id_2.**place goto __)
	emit(goto ___)

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.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 $\rightarrow$ A

{ S.nextlist := nil }

(6) L $\rightarrow$ L_1 M S

{ backpatch(L_1.nextlist , M.quad);
L.nextlist := S.nextlist }

(7) N $\rightarrow$ ε

{ N.nextlist := makelist (nextquad);
emit('goto_') }

(8) M $\rightarrow$ 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.

   $S \rightarrow 0|1|AS0|BS0 \quad A \rightarrow \varepsilon \quad B \rightarrow \varepsilon$ **(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

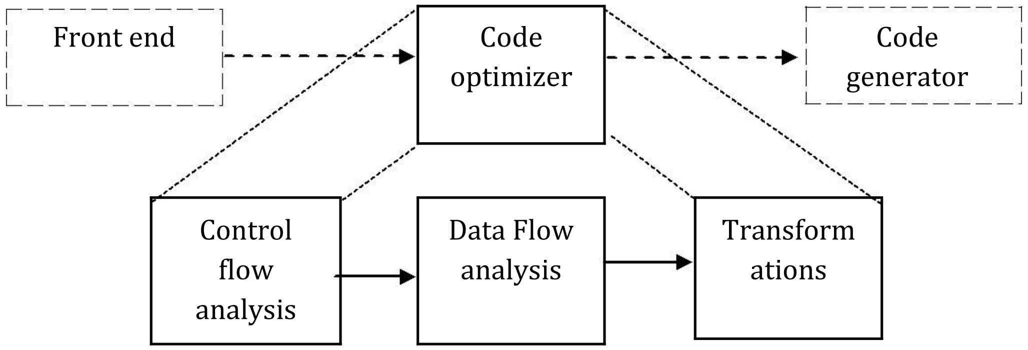
1. What is code optimization?

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 t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. 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)) t_1 := B + A;

t_2 := Y - t_1;

t_3 := t_1 * t_2;

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) $\rightarrow$ 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; $\rightarrow$ 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***

Initial code		Algebraic simplification
A := X**2;		A := X*X;
B := 3;		B := 3;
C := X;		C := X;
D := C*C;	$\Longleftrightarrow$	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 :=

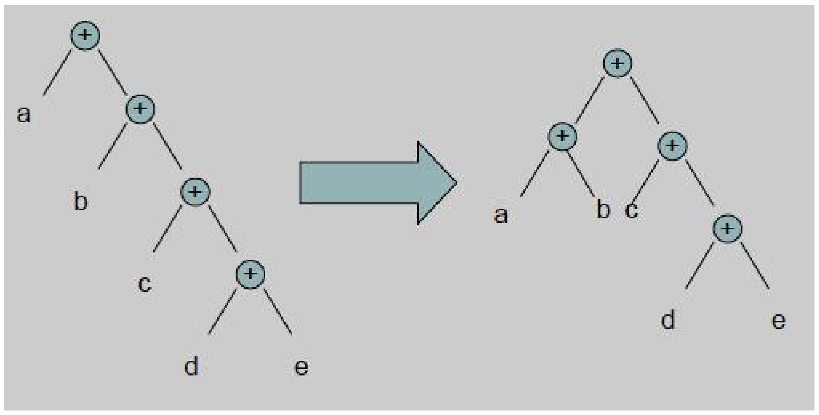
false b || true :=

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

$temp_1=5;$		$temp_1=5;$		$temp_1=15+a;$
$temp_2=temp_1+a;$	$\Rightarrow$	$temp_2=temp_1+a;$	$\Rightarrow$	$b=temp_1;$
$temp_3=temp_2+10;$		$b=temp_1;$		
$b=temp_3;$				

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:

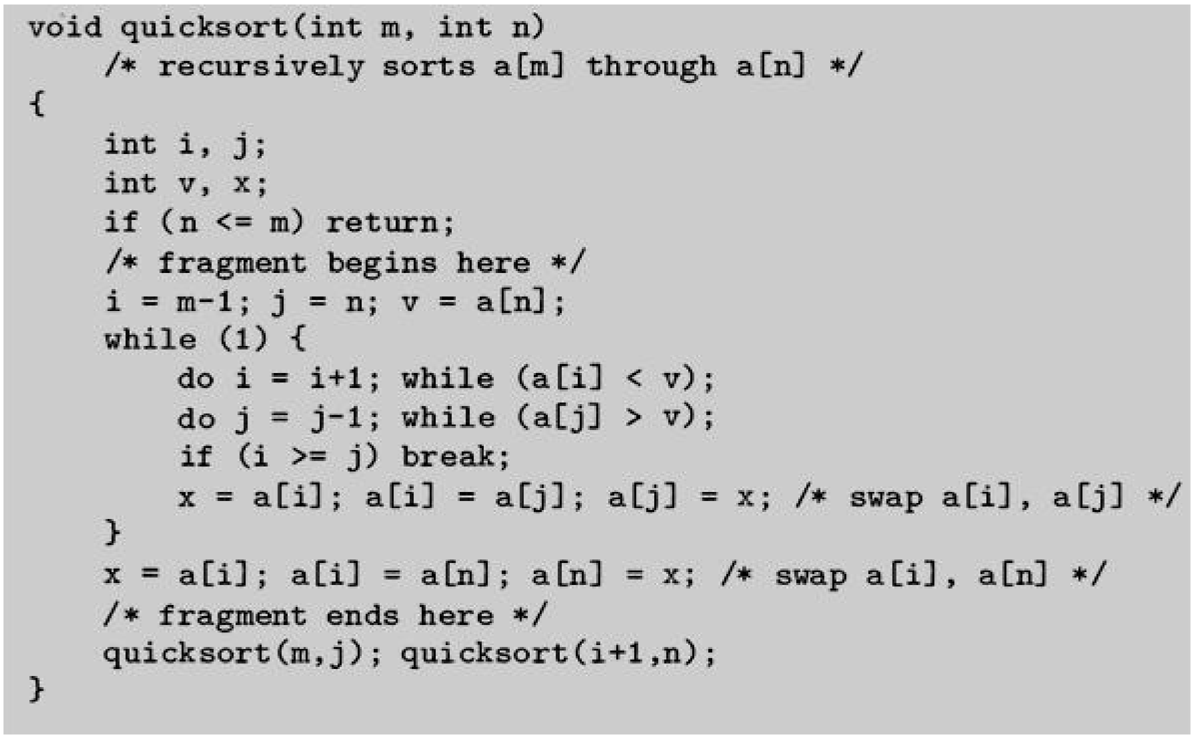
- There are a number of ways in which a compiler can improve a program without changing the function it computes.

The examples of function preserving transformations are

1. Common sub-expression elimination
2. Copy propagation
3. Dead code elimination
4. Constant folding

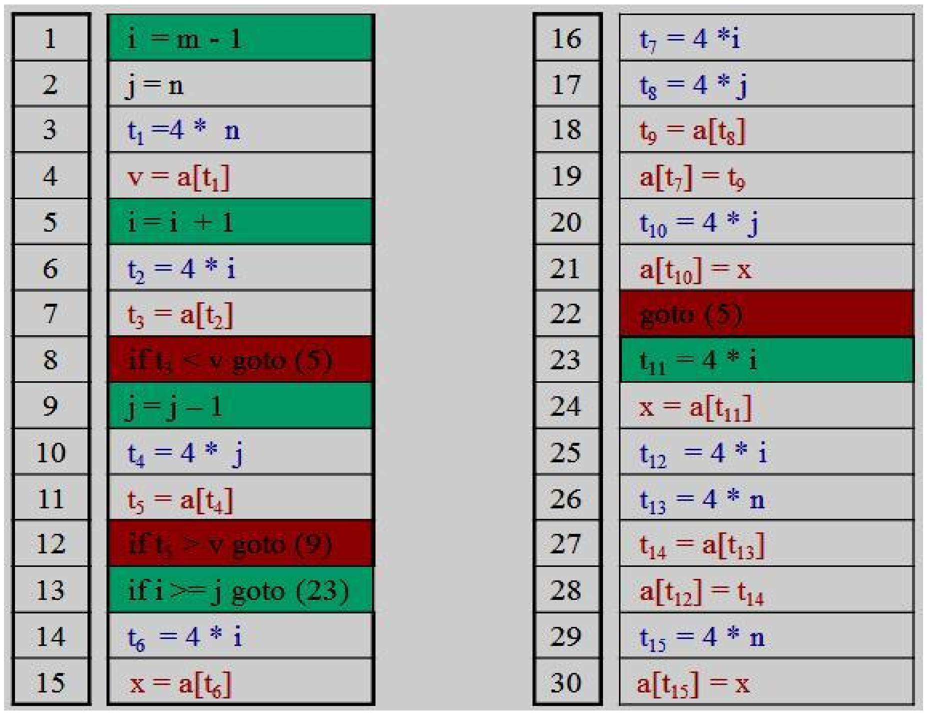
The DAG representation of basic blocks showed how local common sub-expressions could be removed as the DAG for the basic block is constructed.

Quick sort for c program:



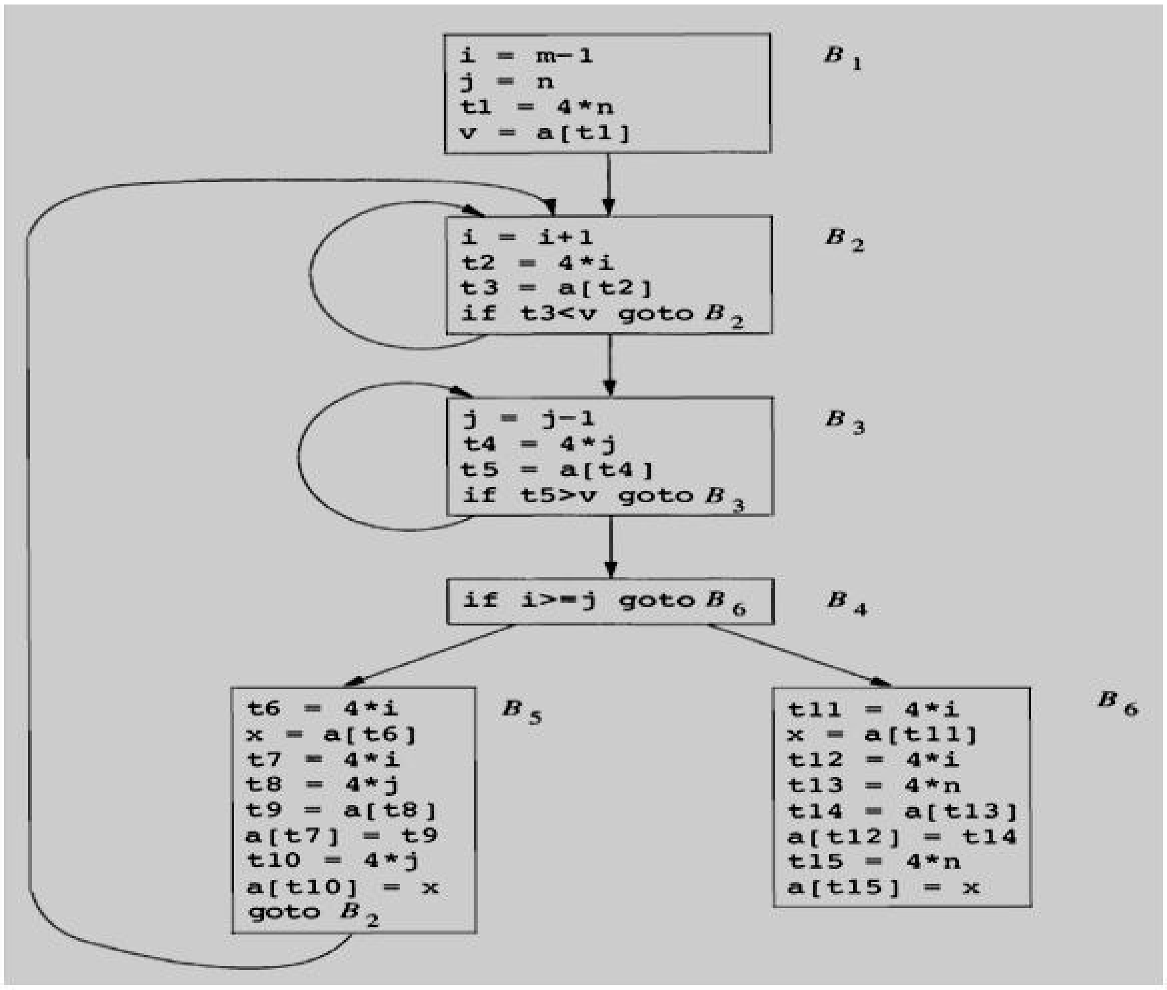
```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= 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
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]
16	t_7 = 4 *i
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * i
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

Flow graph:



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

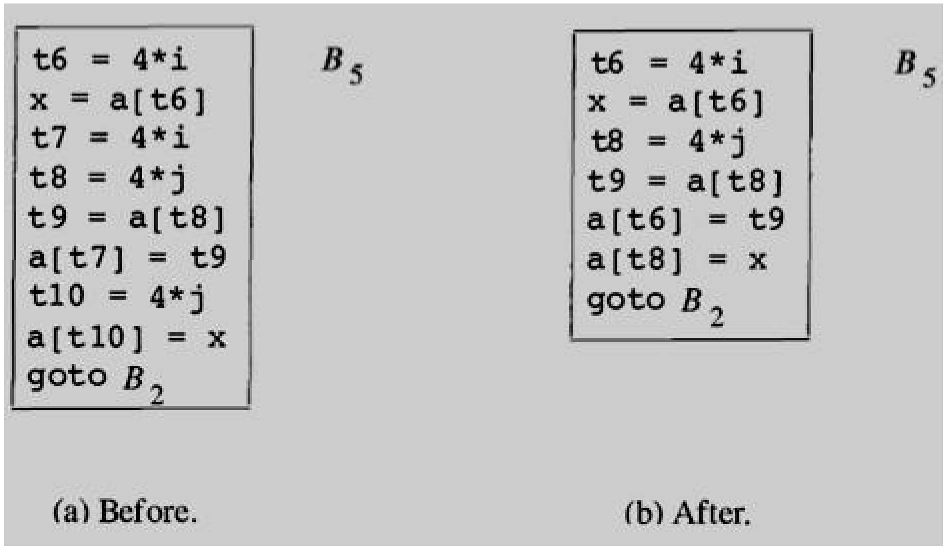
```
if i>=j goto B6
```

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

(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 ***t_7*** and ***t_{10}*** have the common sub-expressions ***4*i*** and ***4*j,*** respectively. They have been eliminated by using ***t_6*** instead of ***t_7*** and ***t_8*** instead of ***t_{10}***.


```
t6 = 4*i            B5
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```
(a) Before.

```
t6 = 4*i            B5
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```
(b) After.

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 B_5 and B_6 in the flow graph.
- After local common sub-expressions are eliminated B_5 still evaluates 4*i and 4*j.
- Both are common sub-expressions; in particular, the three statements

  t_8:= 4*j; t_9:= a [t_8]; a [t_8]:= x in B_5 can be

  replaced by

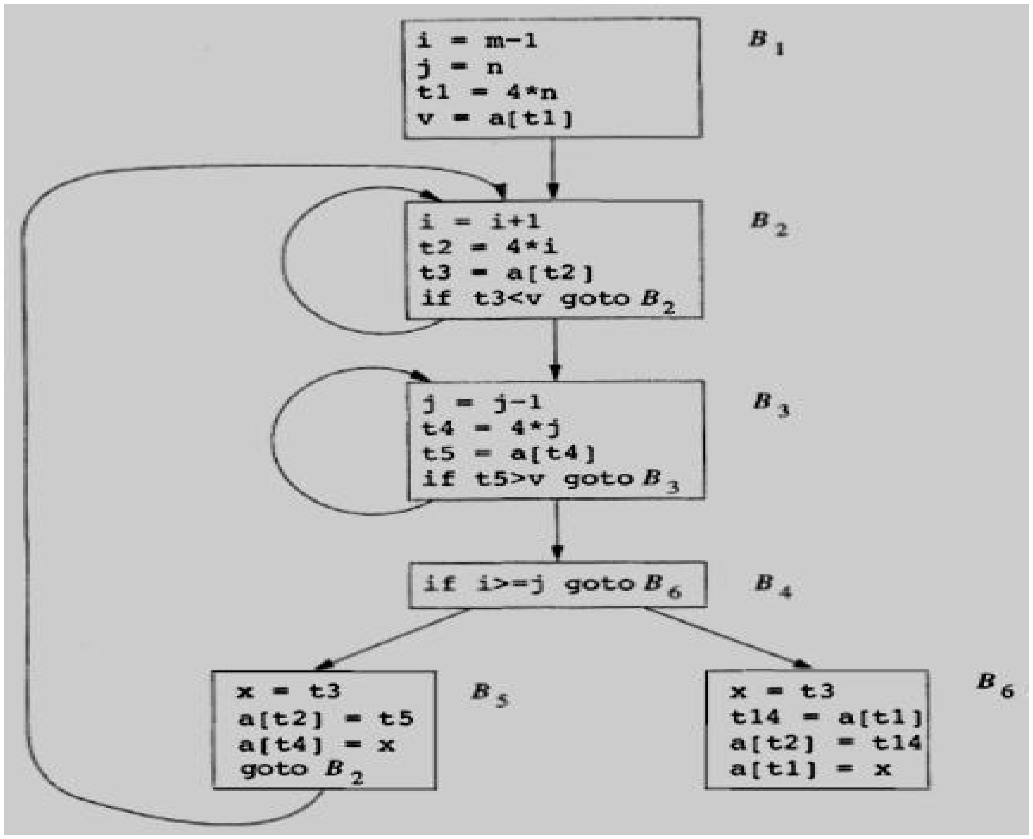
  t_9:= a [t_4]; a [t_4]:= x using t_4 computed in block B_3.
- The control passes from the evaluation of 4*j in B_3 to B_5, there is no change in j, so t_4 can be used if 4*j is needed.
- Another common sub-expression comes to light in B_5 after ***t_4 replaces t_8***.
- The new expression a[t_4] corresponds to the value of a[j] at the source level.

The statement

t_9:= a [t_4]; a [t_6]:= t_9 in B_5 can

therefore be replaced by

a [t_6]:= t_5



Elimination of global and local common Subexpression

(ii) Copy Propagation:

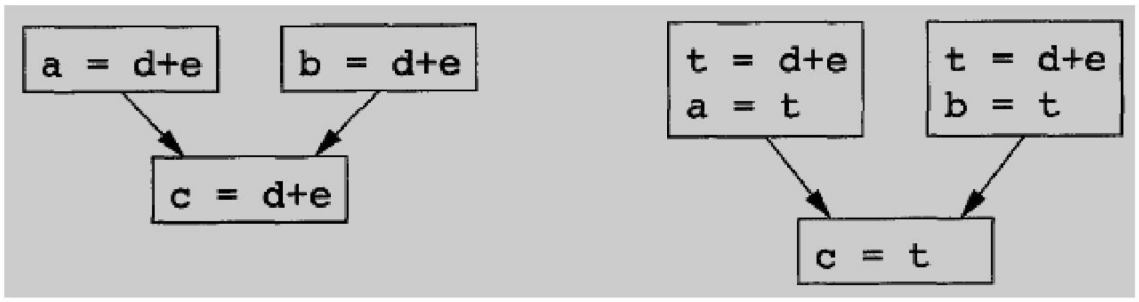
- Block B_5 can be further improved by eliminating x using two new transformations.
- The assignments of the form **f := g** called **copy statements**, or ***copies***
- When the common sub-expression in ***c: = d+e*** is eliminated the algorithm uses a new variable ***t*** to hold the value of ***d+e.***
- 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: = t_3*** in block B_5 is a copy. Copy propagation applied to B_5 yields:

x := t_3

a [t_2]:=t_5

a [t_4]:=t_3

goto B_2



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 [t_2] := t_5

  a [t_4] := t_3

  goto B_2

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

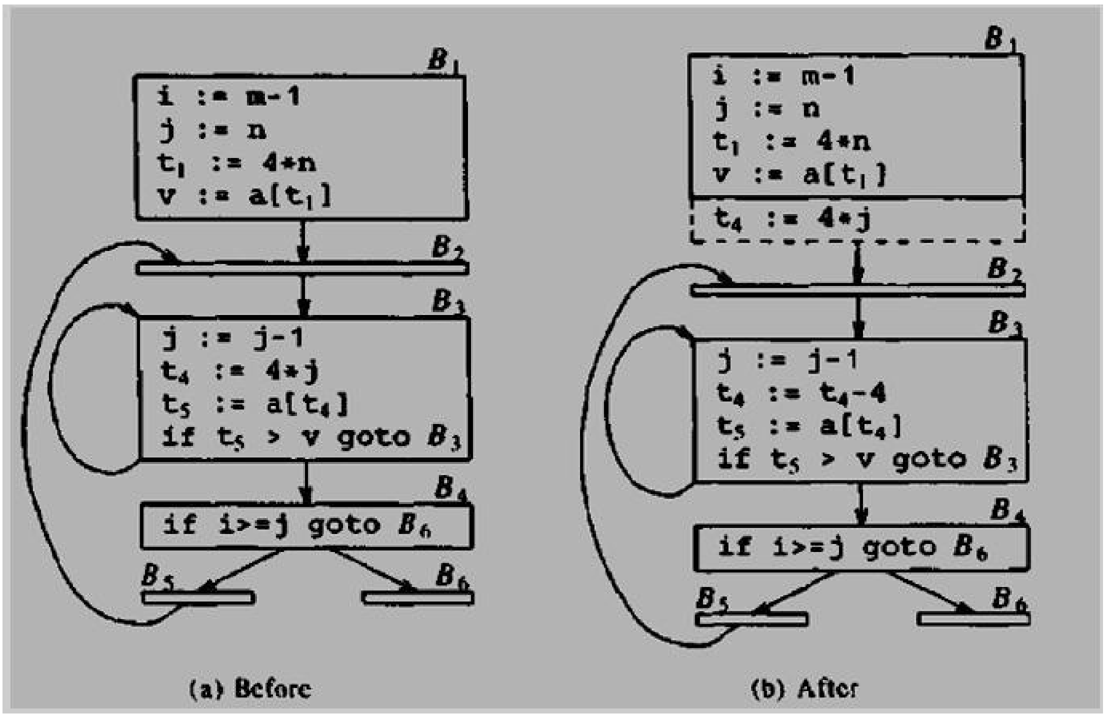
of ***t= limit-2;***

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 t_4 decreases by 4 because 4*j is assigned to t_4.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 B_3, t_4 is used in B_3 and j in B_4.

Strength reduction applied to 4*j in block B_3

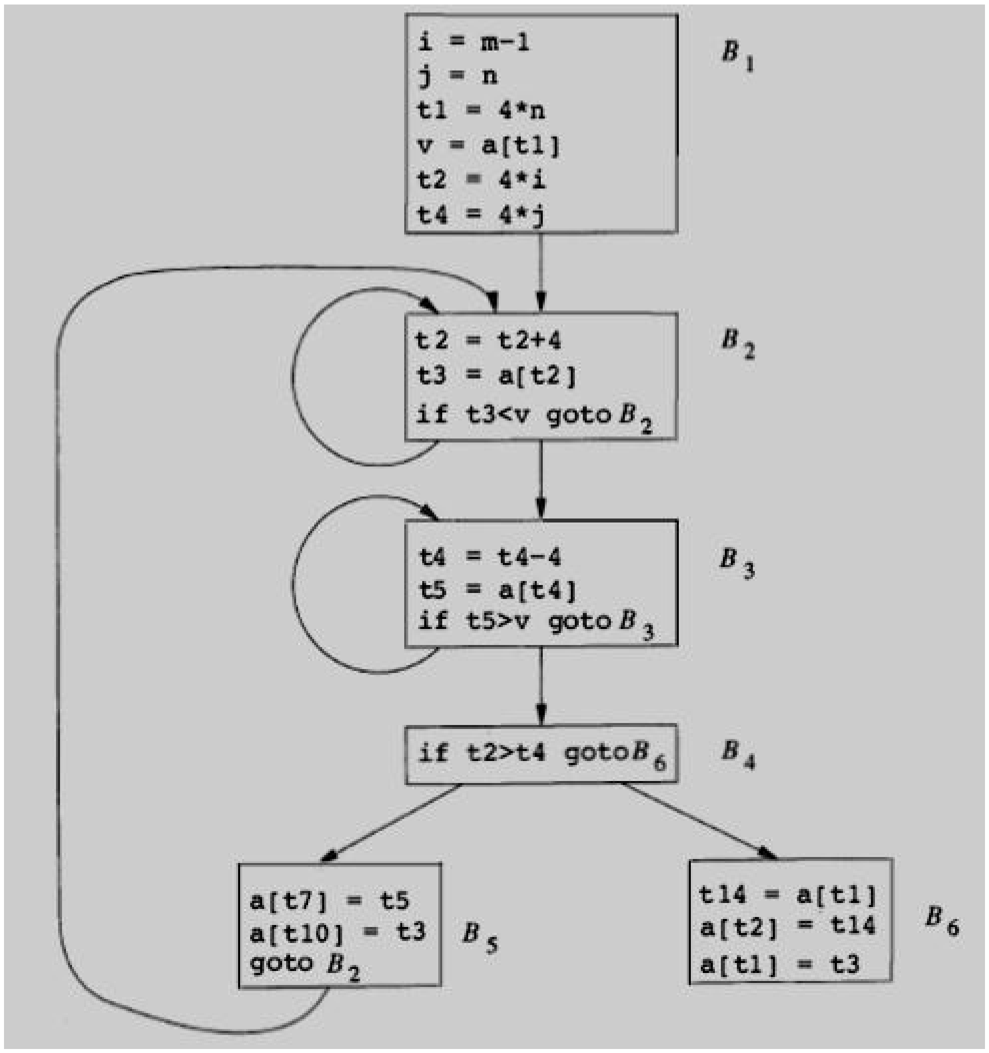


- In block B_3, t_4 := 4 * j holds assignment to t_4 and t_4 is not changed elsewhere in the inner loop around B_3.
- It follows that the statement j := j - 1 the relationship t_4 := 4 *j - 4 must hold.
- The replace the assignment **t_4 := 4 * j** by **t_4 := t_4 - 4.**
- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction.

(iii) Reduction in Strength:

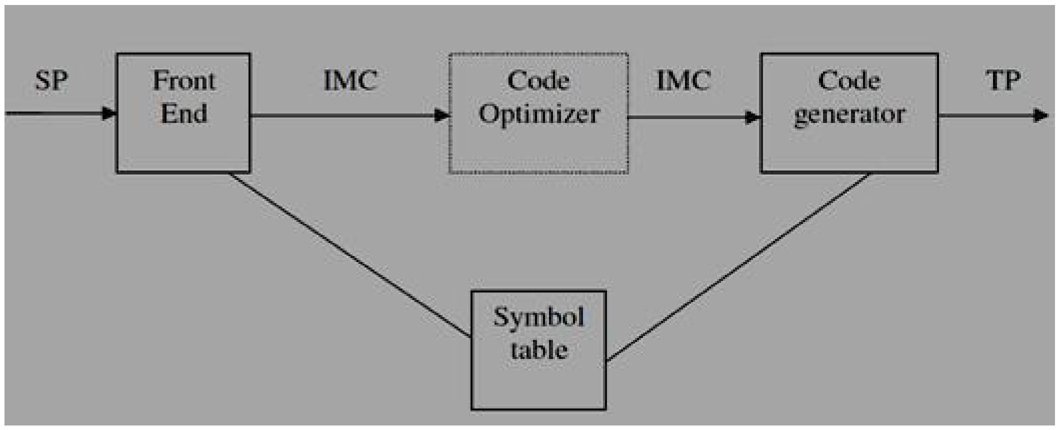
- **Reduction in strength,** which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.
- After reduction in strength is applied to the inner loop around B_2 and B_3, the only use in i and j to determine the outcome of test in block B_4.
- The value of **i** and **t_2 = 4 * i** and **j** for **t_4 = 4 * j** , so the test **t_2 >= t_4** is equivalent to i >= j.
- Replacement i in block B_2 and j in Block B_3 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.

(a)	(b)
t := a + b	**t := a + b**
t := t * c	**t := t + c**
t := t / d	**t := t / d**

Two three address code sequences

(a)	(b)
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

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, R_0,R_1,…., R_{n-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 **cost = 6**
MOV R0, a

2. MOV b, a **cost = 6**
ADD c, a

Assume R0 = a, R1 = b, R2 = c, respectively

3. MOV *R1,*R0 **cost = 2**
ADD *R2,*R0

Assume R1 = b, R2 = c

4. ADD R1, R2 **cost = 3**
MOV R1, a

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

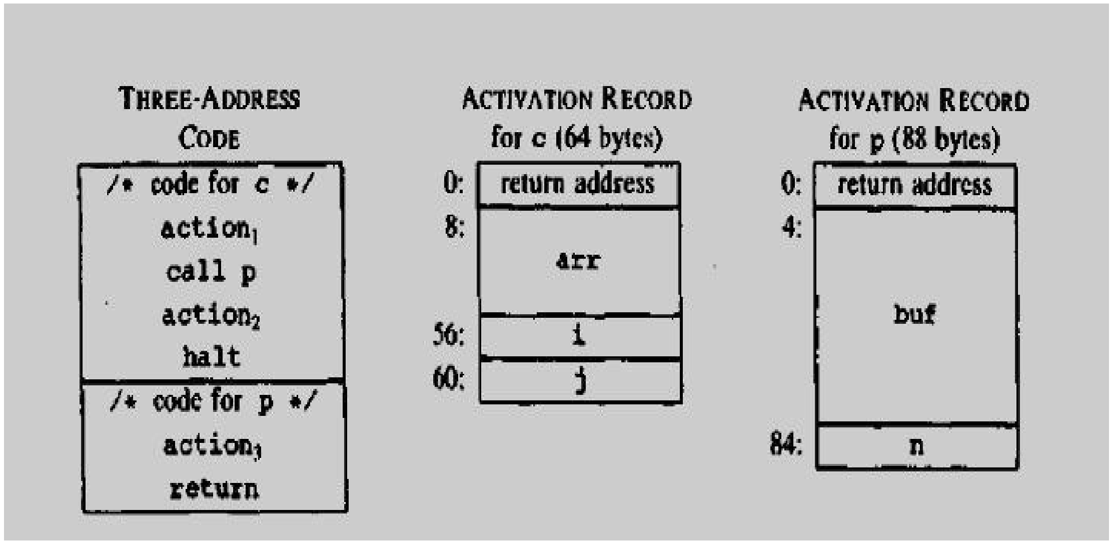
- The ***callee.static_area*** and ***callee.code_area*** are constants referring to the address of the activation record and the first instruction for the called procedure respectively.
- The source **#here+20** in the **MOV** instruction is the literal return address.
- The code for a procedure ends 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

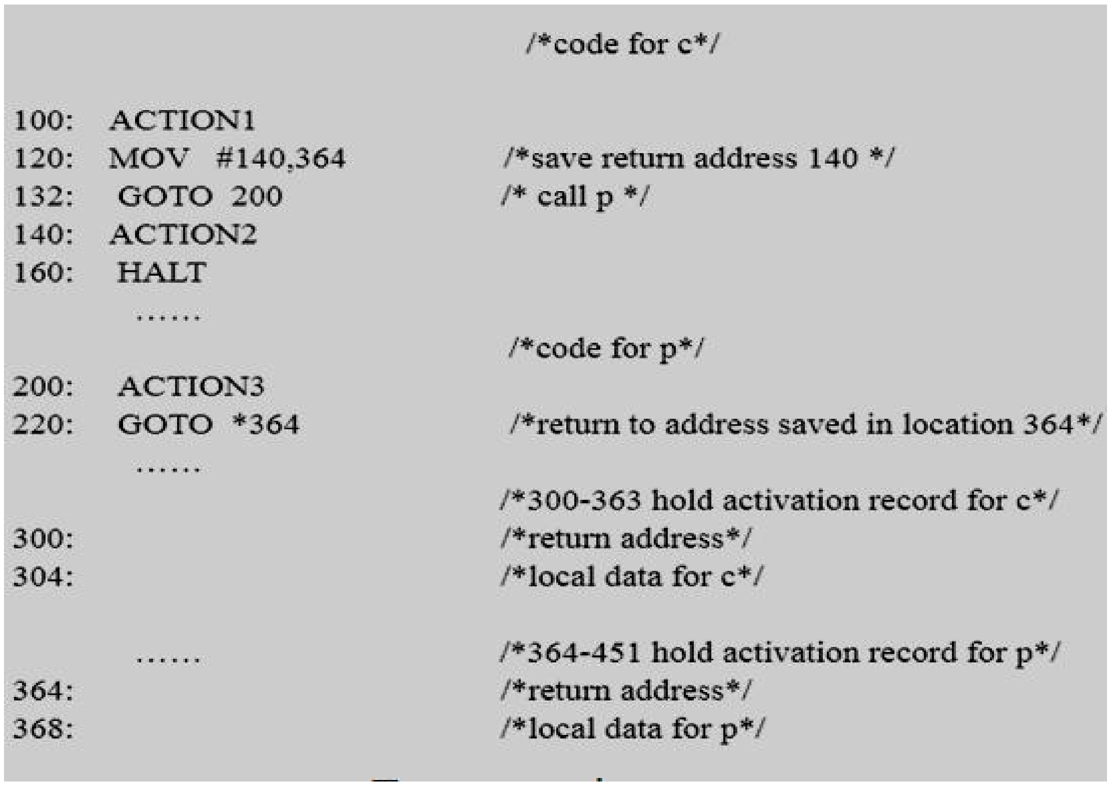
GOTO *callee.code_area

- 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:

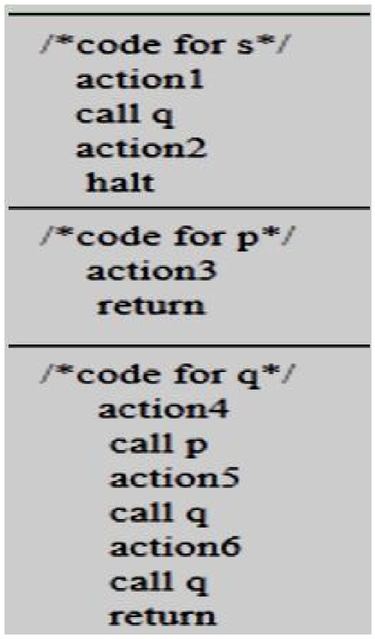
- 0(SP) is the address of the first word in the activation record and
- *0(SP) is the return address saved there.

- 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
```

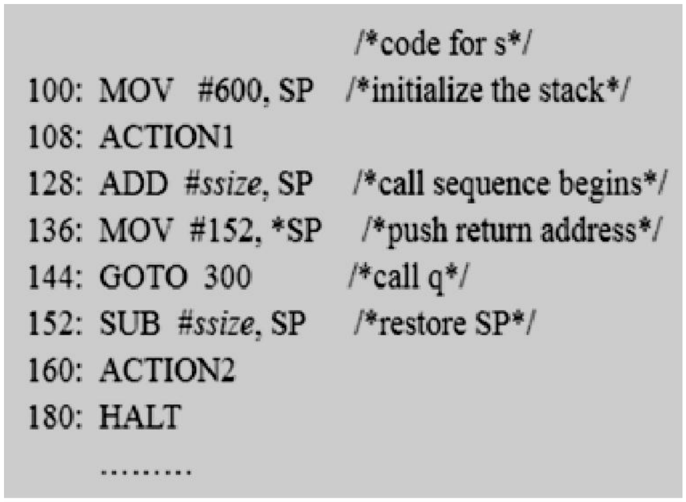
Target code for stack allocation:

Three-address code

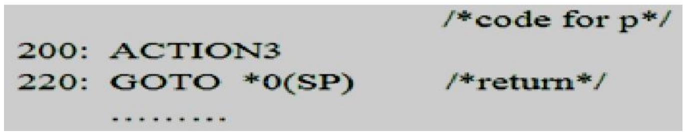


```
/*code for s*/
   action1
   call q
   action2
    halt

/*code for p*/
    action3
     return

/*code for q*/
     action4
      call p
      action5
      call q
      action6
      call q
      return
```

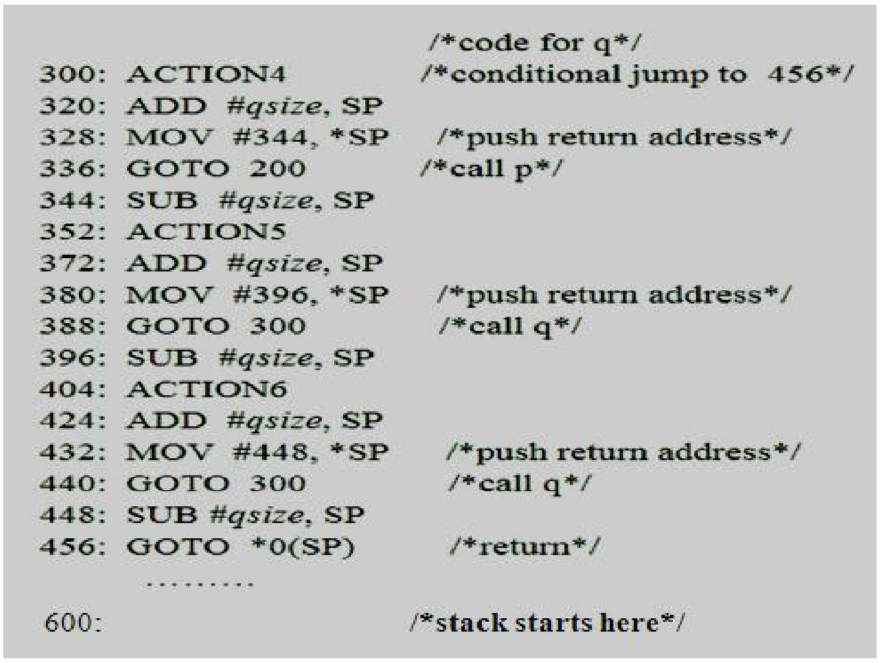
Three address code for stack allocation:



```
                          /*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

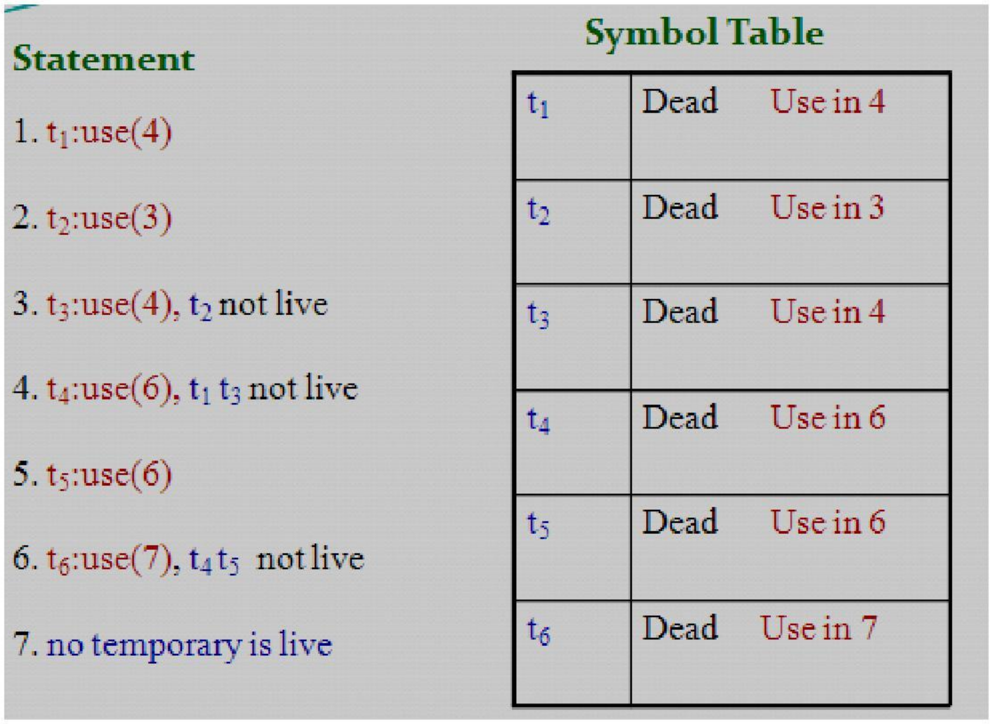
1. Attach to statement i, information currently found in the symbol table regarding the next use and live-ness of X, Y and Z.
2. In symbol table, set X to “not live” and “no next use “.
3. In symbol table, set Y and Z to be “live” and next use of Y and Z to i.

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 t_1 and t_2:

Example: X = a * a + 2(a * b) + (b * b)

1. t_1 = a * a
2. t_2 = a * b
3. t_3 = 2 * t_2
4. t_4 = t_1 + t_3
5. t_5 = b * b
6. t_6 = t_4 + t_5
7. X = t_6



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
```

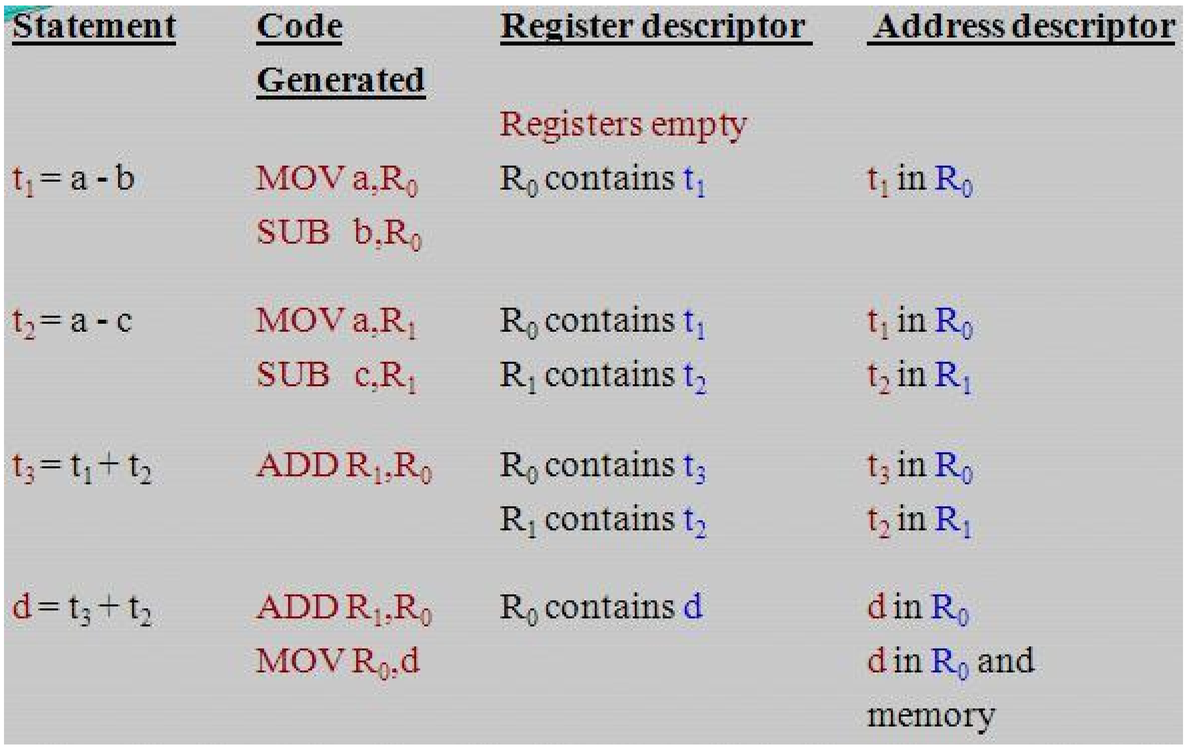
The *getreg ()* function:

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.

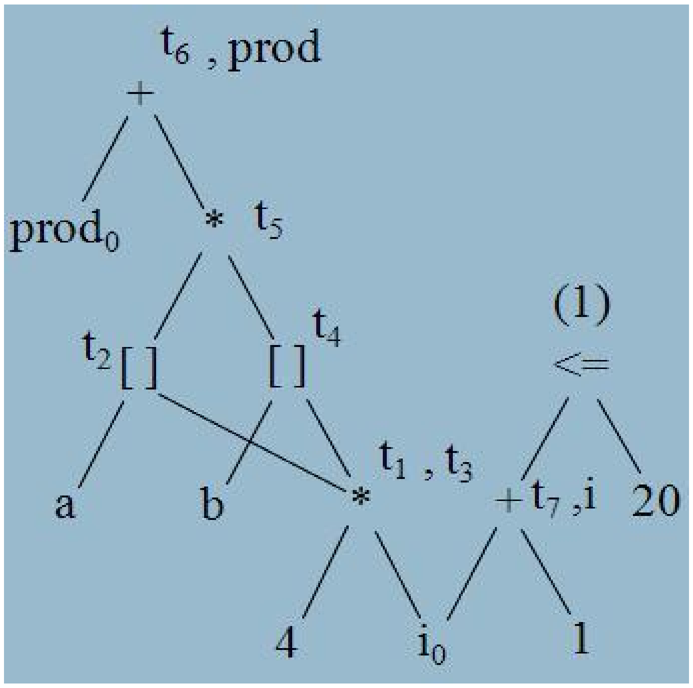
The source program

```
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. t_1 := 4 * i
2. t_2 := a[t_1]
3. t_3 := 4 * i
4. t_4 := b[t_3]
5. t_5 := t_2 * t_4
6. t_6 := prod + t_5
7. prod := t_6
8. t_7 := i + 1
9. i := t_7
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, and ***node*** is undefined for all arguments. Suppose the current three address statements in either **(i) x:= y op z (ii) x:= op y (iii) x:= y.** A relational operator like if i<=20 goto case (i), with X undefined.

(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, R_0**

- We can delete instruction (2) because whenever (2) is executed.
- Instruction (1) the value of a is already in register R_0.

(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 sequence

```
            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:

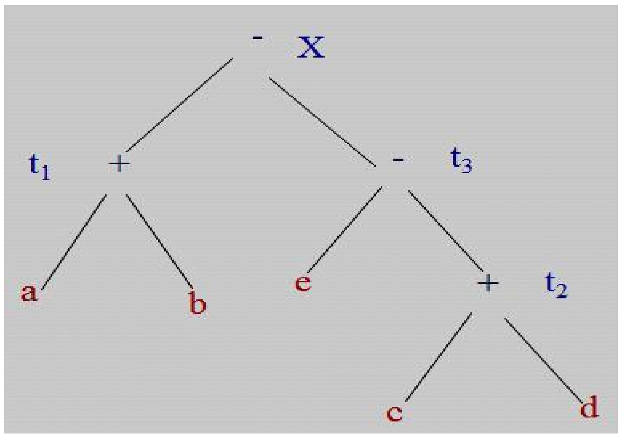
- Consider how the order in which computations are done can affect the cost of resulting object code.
- Consider the following basic block whose dag representation

$t_1 := a + b$

$t_2 := c + d$

$t_3 := e - t_2$

$X := t_1 - t_3$



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, R_0

ADD b, R_0

MOV c, R_1

ADD d, R_1

MOV R_0, t_1

MOV e, R_0

SUB R_1, R_0

MOV t_1, R_1

SUB R_0, R_1

MOV R_1, X

We rearranged the order of the statements so that the computation of t_1 occurs immediatedly before that of t_4 as:

```
t2 := c + d
t3 := e −t2
t1 := a + b
X := t1 – t3
```

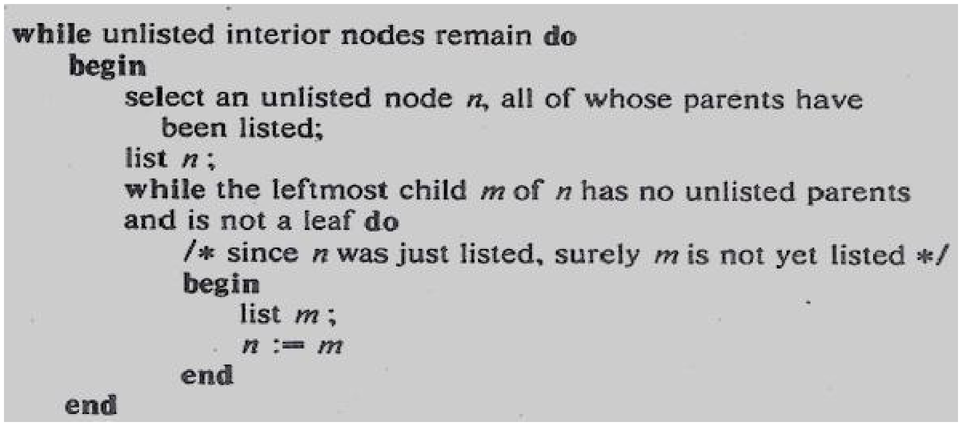
Using the code generation algorithm, the code sequence as

```
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:

$t_8 := d + e$

$t_6 := a + b$

$t_5 := t_6 - c$

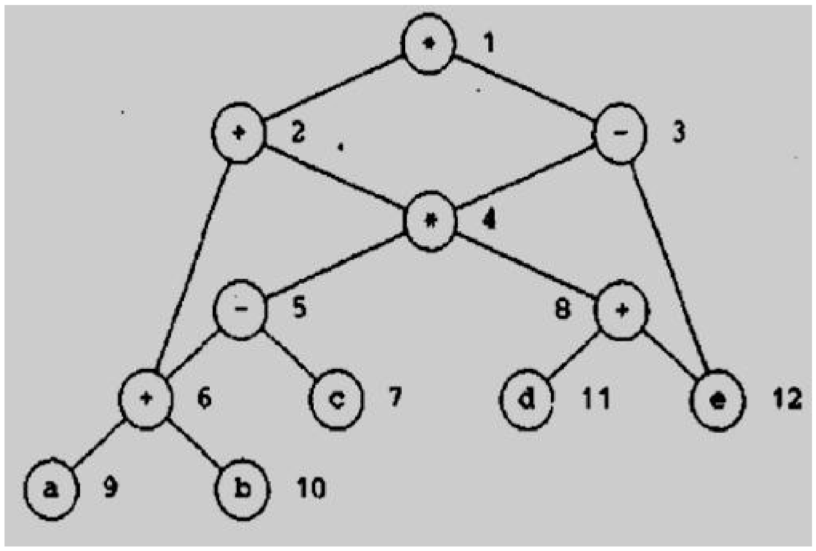
$t_4 := t_5 * t_8$

$t_3 := t_4 - e$

$t_2 := t_6 + t_4$

$t_1 := t_2 * t_3$

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.

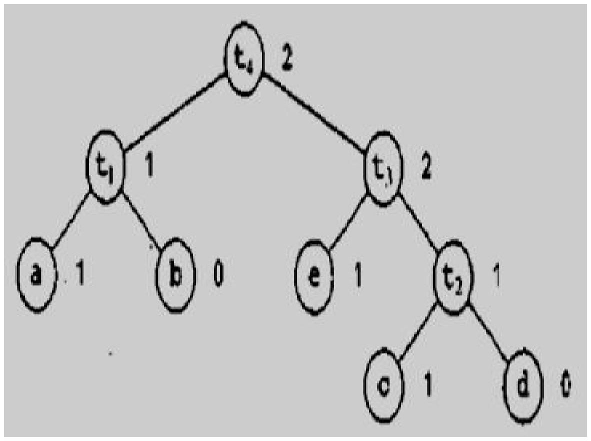
The important ***n*** is a ***binary node*** and its children have labels l_1 and l_2,

$$\textbf{label(n)} = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1+1 & \text{if } l_1 = l_2 \end{cases}$$

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 n_1, n_2, . . . , n_k be the children of n ordered by LABEL,
          so LABEL(n_1) ≥ LABEL(n_2) ≥ · · · ≥ LABEL(n_k);
        LABEL(n) := max_{1≤i≤k} (LABEL(n_i)+i−1)
    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:

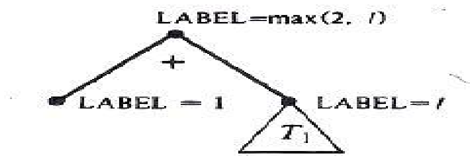
- The labeling algorithm to handle operations like multiplication, division or a function call, which normally require more than one register to perform.
- The labeling algorithm ***label (n)*** is always at least the number of registers required by the operation.
- If multiplication requires two registers, in the binary case use

$$\textbf{label (n)} = \begin{cases} \max(2, l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1+1 & \text{if } l_1 = l_2 \end{cases}$$

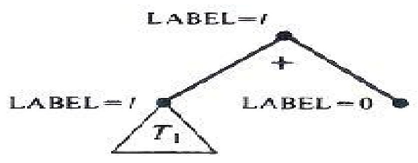
- where l_1 and l_2 are the labels of the children of n.

(v) Algebraic Properties:

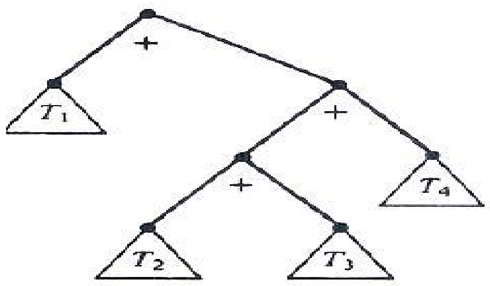
- The algebraic laws for various operators, the opportunity to replace a given tree T by one with smaller labels and fewer left leaves.



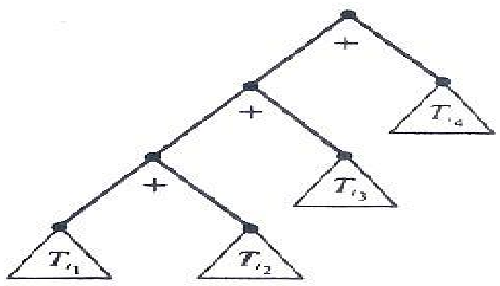
(a)



(b)



(c)



(d)

(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)**