VTU question paper solution -SS and CD -18CS61, July 2022

MODULE-1

1

a) Explain in details SIC-XE machine architecture -10 marks

Solution:

1. Memory

- Maximum memory available on a SIC/XE system is 1 megabyte (2^{20} bytes)
- An address (20 bits) cannot be fitted into a 15-bit field as in SIC Standard
- Must change instruction formats and addressing modes

2. Registers

- **9 registers (5 registers of SIC + 4 additional registers)**

Mnemonic	Number	Special use
A	0	Accumulator
X	1	Index register
L	2	Linkage register
PC	8	Program counter
SW	9	Status word

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

3. Data Format

- 24-bit(3 Bytes) integer representation in 2's complement
- 8-bit(1 Byte) ASCII code for characters

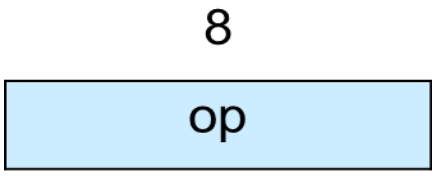
There is a 48-bit floating-point data type

- *fraction* is a value between 0 and 1
- *exponent* is an unsigned binary number between 0 and 2047

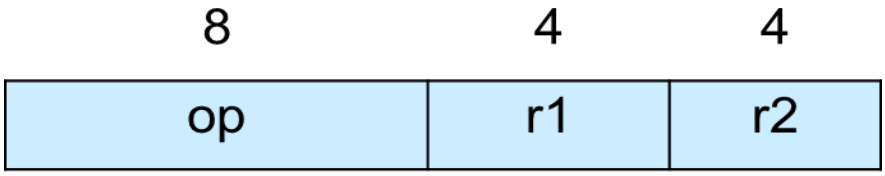
- zero is represented as all 0

4. **Instruction Format**

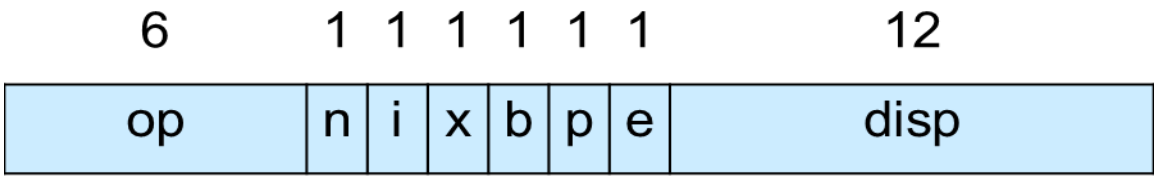
Format 1(1 Byte)



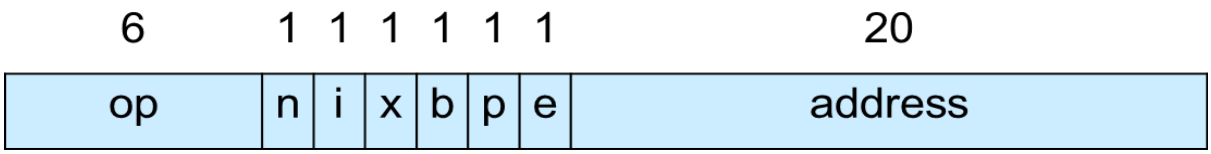
Format 2(2 Byte)



Format 3(3 Byte)

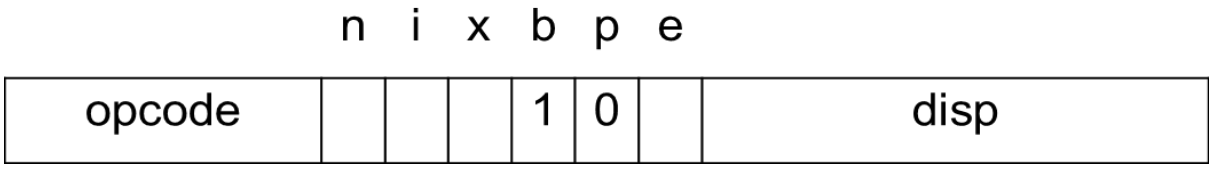


Format 4(4 Byte)



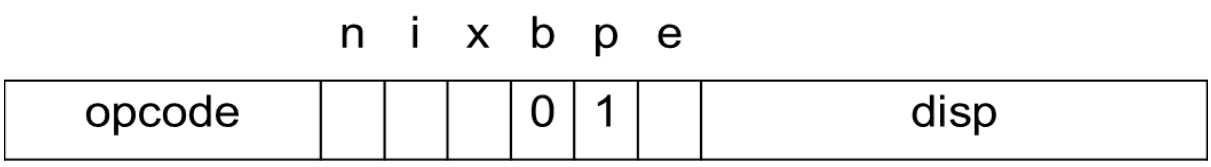
5. **Addressing Mode**

Base Relative Addressing Mode



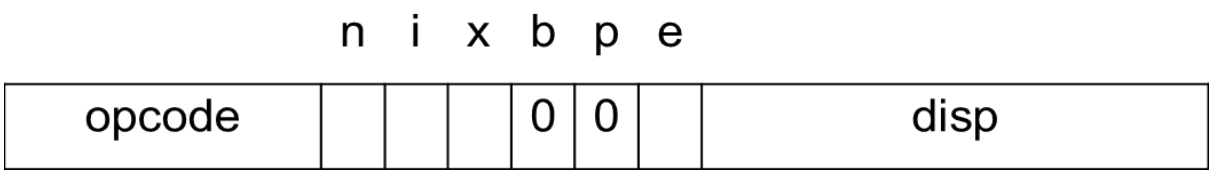
b=1, p=0, TA=(B)+disp (0≤disp≤4095)

Program-Counter Relative Addressing Mode



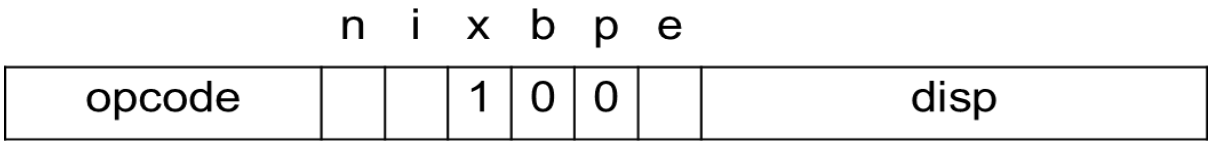
b=0, p=1, TA=(PC)+disp (-2048≤disp≤2047)

Direct Addressing Mode



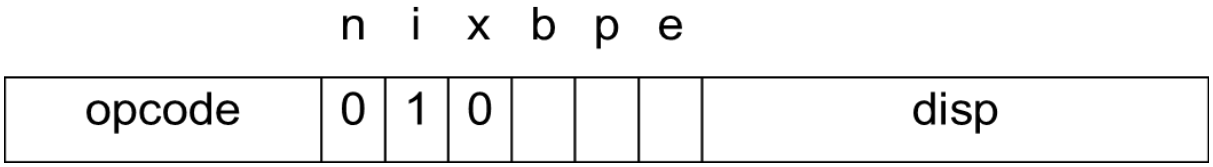
b=0, p=0, TA=disp (0≤disp≤4095)

Index Addressing Mode



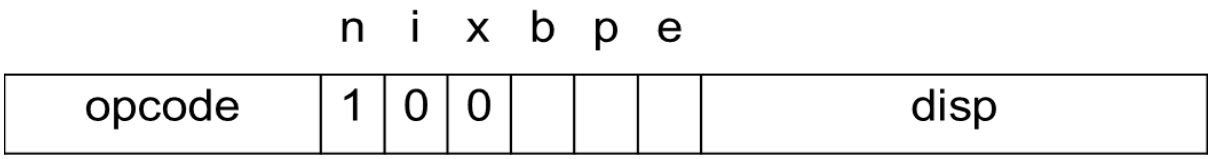
b=0, p=0, TA=(X)+disp

Immediate Addressing Mode



n=0, i=1, x=0, operand=disp

Indirect Addressing Mode



n=1, i=0, x=0, TA=(disp)

6. **Instruction Set**
   - load and store:LDA, LDX, STA, STX LDB, STB, etc.
- Integer arithmetic: ADD, SUB, MUL, DIV
   - Floating-point arithmetic operations
     – ADDF, SUBF, MULF, DIVF
   - Register move: RMO
   - Register-to-register arithmetic operations

- ADDR, SUBR, MULR, DIVR
- Supervisor call: SVC

- Conditional jump instructions
  - JLT, JEQ, JGT: test CC and jump
- Subroutine linkage
  - JSUB, RSUB: return address in register L
- Input and output
  - Performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A
  - Each device is assigned a unique 8-bit code, as an operand of I/O instructions
  - Test Device (TD): < (ready), = (not ready)
  - Read Data (RD), Write Data (WD)

- Input and output:
  - I/O channels to perform I/O while CPU is executing other instructions: SIO, TIO, HIO

b) List the various machine independent assembler features .explain the control sections how the assembler convert them into object code. 10 marks

Solution:

Machine-Independent Assembler Features are

1. Literals
2. Symbol-Defining Statements
3. Expressions
4. Program Blocks
5. Control Sections
6. Program Linking

Control Sections

A control section is a part of the program that maintains its identity after assembly; each such control section can be loaded and relocated independently of the others.

Different control sections are most often used for subroutines or other logical subdivisions of a program. Control sections differ from program blocks in that they are handled separately by the assembler.

- The EXTDEF (external definition) statement in a control section names symbols, called external symbols that are defined in this control section and may be used by other sections.
- The EXTREF (external reference) statement names symbols that are used in this control section and are defined elsewhere. We need two new record types (Define and Refer) in the object program.

• A Define record gives information about external symbols that are defined in this control section – that is, symbols named by EXTDEF.
• A Refer record lists symbols that are used as external reference by the control section – that is, symbols named by EXTREF.

Handling External Reference

Case 1

15 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 formatto provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loaderto perform the required linking

Case 2

190 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

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

- 107 1000 MAXLEN EQU BUFEND-BUFFER

Object Code for the example program:

0000	COPY	START	0	
		EXTDEF	BUFFER,BUFFEND,LENGTH	
		EXTREF	RDREC,WRREC	
0000	FIRST	STL	RETADR	172027
0003	CLOOP	+JSUB	RDREC	4B100000
0007		LDA	LENGTH	032023
000A		COMP	#0	290000
000D		JEQ	ENDFIL	332007
0010		+JSUB	WRREC	4B100000
0014		J	CLOOP	3F2FEC
0017	ENDFIL	LDA	=C'EOF'	032016
001A		STA	BUFFER	0F2016
001D		LDA	#3	010003
0020		STA	LENGTH	0F200A
0023		+JSUB	WRREC	4B100000
0027		J	@RETADR	3E2000
002A	RETADR	RESW	1	
002D	LENGTH	RESW	1	
		LTORG		
0030	*	=C'EOF'		454F46
0033	BUFFER	RESB	4096	
1033	BUFEND	EQU	*	
1000	MAXLEN	EQU	BUFEND-BUFFER	

Case 1

```
0000    RDREC   CSECT
        .
        .       SUBROUTINE TO READ RECORD INTO BUFFER
        .
                EXTREF  BUFFER,LENGTH,BUFEND
0000            CLEAR   X               B410
0002            CLEAR   A               B400
0004            CLEAR   S               B440
0006            LDT     MAXLEN          77201F
0009    RLOOP   TD      INPUT           E3201B
000C            JEQ     RLOOP           332FFA
000F            RD      INPUT           DB2015
0012            COMPR   A,S             A004
0014            JEQ     EXIT            332009
0017            +STCH   BUFFER,X        57900000
001B            TIXR    T               B850
001D            JLT     RLOOP           3B2FE9
0020    EXIT    +STX    LENGTH          13100000
0024            RSUB                    4F0000
0027    INPUT   BYTE    X'F1'           F1
0028    MAXLEN  WORD    BUFFEND-BUFFER  000000      Case 2
```

```
0000    WRREC   CSECT
        .
        .       SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
                EXTREF  LENGTH,BUFFER
0000            CLEAR   X               B410
0002            +LDT    LENGTH          77100000
0006    WLOOP   TD      =X'05'          E32012
0009            JEQ     WLOOP           332FFA
000C            +LDCH   BUFFER,X        53900000
0010            WD      =X'05'          DF2008
0013            TIXR    T               B850
0015            JLT     WLOOP           3B2FEE
0018            RSUB                    4F0000
                END     FIRST
0018    *       =X'05'                  05
```

2 a. Write an algorithm for One Pass Assembler and give sample object program from One Pass Assembler. **(10 Marks)**

Algorithm:

Begin
read first input line
if OPCODE = ‘START’ then begin
save #[Operand] as starting addr
initialize LOCCTR to starting address
write line to intermediate file
read next line
end(if START)
else
initialize LOCCTR to 0
While OPCODE != ‘END’ do
begin
if this is not a comment line then
begin
if there is a symbol in the LABEL field then

```
begin
search SYMTAB for LABEL
if found then
set error flag (duplicate symbol)
else
(if symbol)
search OPTAB for OPCODE
if found then
add 3 (instr length) to LOCCTR
else if OPCODE = 'WORD' then
add 3 to LOCCTR
else if OPCODE = 'RESW' then
add 3 * #[OPERAND] to LOCCTR
else if OPCODE = 'RESB' then
add #[OPERAND] to LOCCTR
else if OPCODE = 'BYTE' then
begin
find length of constant in bytes
add length to LOCCTR
end
else
set error flag (invalid operation code)
end (if not a comment)
write line to intermediate file
read next input line
end { while not END}
write last line to intermediate file
Save (LOCCTR – starting address) as
program length
End {pass 1}
```

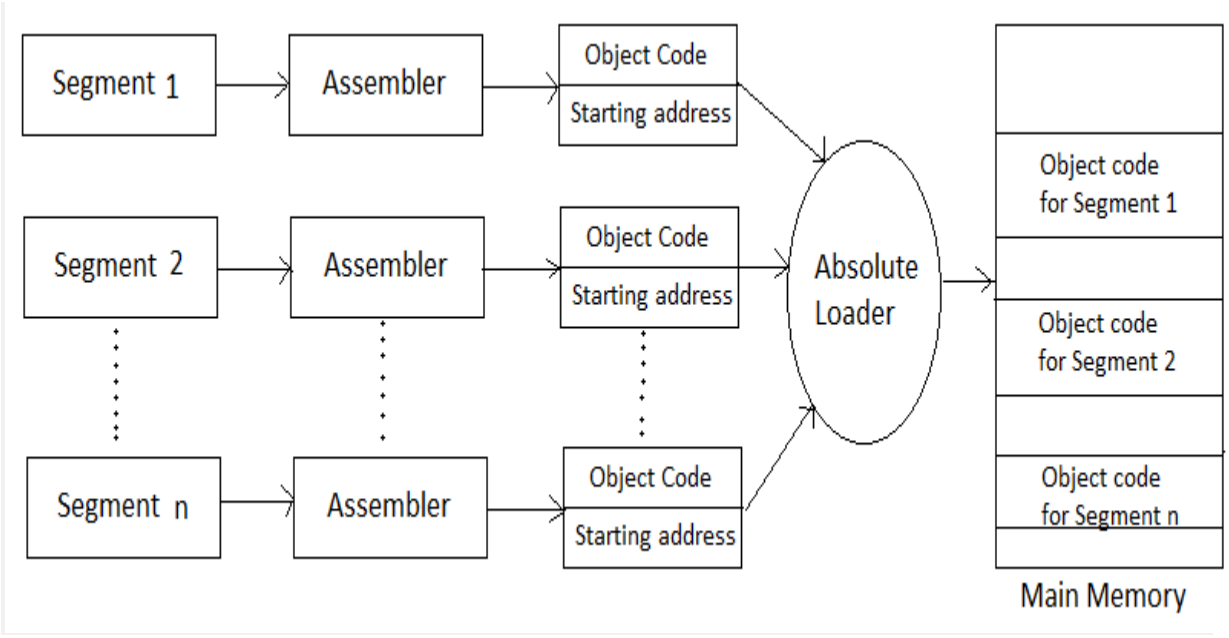
b. What are the basic functions of loader? Explain two ways of program relocation in loaders. **(10 Marks)**

Solution:

Basic function of loader are

1. **Allocation:** It allocates memory for the program in the main memory.
2. **Linking:** It combines two or more separate object programs or modules and supplies necessary information.
3. **Relocation:** It modifies the object program so that it can be loaded at an address different from the location.
4. **Loading:** It brings the object program into the main memory for execution.
   - The absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at a specified location in the memory.
   - This type of loader is called absolute loader because no relocating information is needed, rather it is obtained from the programmer or assembler.
   - The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file then the task of loader becomes very simple that is to simply place the executable form of the machine instructions at the locations mentioned in the object file.

- In this scheme, the programmer or assembler should have knowledge of memory management. The programmer should take care of two things:
- Specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next modules, it's then the programmer's duty to make necessary changes in the starting address of respective modules.
- While branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.



Process of Absolute Loader

RELOCATABLE LOADERS

Absolute loaders have a number of advantages: they are small, fast and simple. But they have a number of disadvantages, too.

The major problem deals with the need to assemble an entire program all at once. Since the addresses for the program are determined at assembly time, the entire program must be assembled at one time in order for proper addresses to be assigned to the different parts. This means that a small change to one subroutine requires reassembly of the entire program. Also, standard subroutines, which might be kept in a library of useful subroutines and functions, must be physically copied and added to each program which uses them.

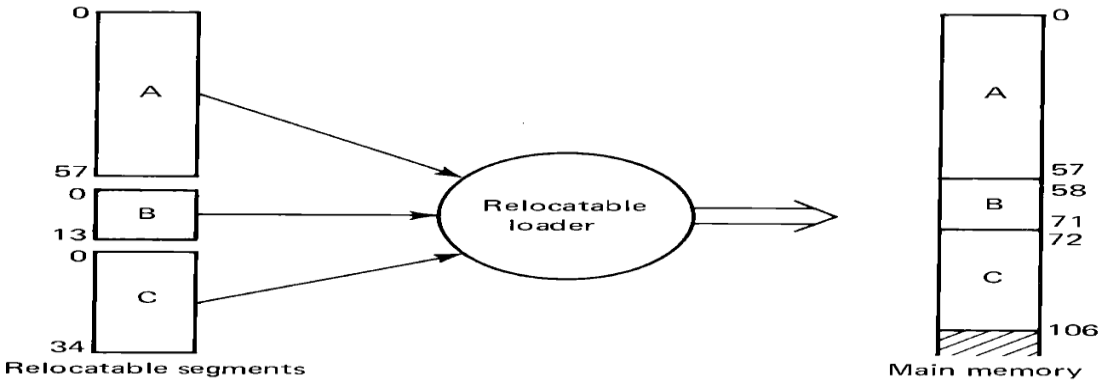
A *relocatable loader* is a loader which allows this delay of binding time. A relocatable loader accepts as input a sequence of segments, each in a special relocatable load format, and loads these segments into memory. The addresses into which segments are loaded are determined by the relocatable loader, not by the assembler or the programmer.

Each segment is a subroutine, function, main program, block of global data, or some similar set of memory locations which the programmer wishes to group together. Segments are

loaded into memory one after the other, to use as little space as possible. The relocatable load format is defined so that separate segments can be assembled or compiled separately and combined at load time.

Relocation

The relocation implied in the name "relocatable loader" refers to the fact that on two separate loads, the same segment can be loaded into two different locations in memory. If any of the segments which are loaded into memory before a segment change in size due to recoding and reassembly between the two loads, then the addresses in memory into which the segment is loaded will change by the same amount.



Consider the following simple program

```
BEGIN     LD2  LENGTH
LOOP      IN   BUFFER(16)
          OUT  BUFFER(18)
          DEC2 1
          J2P  LOOP
          HLT
LENGTH    CON  10
BUFFER    ORIG *+24
          END  BEGIN
```

This program has four symbols, BEGIN, LOOP, LENGTH, and BUFFER. If the program were to be loaded into memory starting at location 0, then the values of these symbols would be 0, 1, 6, and 7, respectively. If the starting address were 1000, the values of the symbols would be 1000, 1001, 1006, and 1007; if the base address were 1976, the values would be 1976, 1977, 1982, and 1983. In all cases, the addresses, for a base BASE, would be BASE+0, BASE+1, BASE+6, and BASE+7. Thus, to relocate the program from starting at an address BASE to starting at an address NEWBASE merely involves adding NEWBASE-BASE to the values of all of the symbols. If the assembler would produce all code as if it had a base of 0, then relocating this code would involve only adding the correct base.

If we were to start the program at 1000, the addresses would be

code address generated

```
BEGIN    LD2  LENGTH          1006
LOOP     IN   BUFFER(16)      1007
         OUT  BUFFER(18)      1007
         DEC2 1                  1
         J2P  LOOP            1001
         HLT                     0
LENGTH   CON  10                10
BUFFER   ORIG *+24
         END  BEGIN           1000
```

3 a. Explain various phases of Compiler. Show the translations for an Assignment statement. Position = Initial + rate * 60. Clearly indicate the output of each phase. **(12 Marks)**

Solution:

. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation

Symbol Table – It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.
The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

1. **Lexical Analyzer** –

   It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

2. **Syntax Analyzer** – It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

3. **Semantic Analyzer** – It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.

4. **Intermediate Code Generator** – It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.
   Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

5. **Code Optimizer** – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.

6. **Target Code Generator** – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The ptimized code is converted into relocatable machine code which then forms the input to the linker and loader.

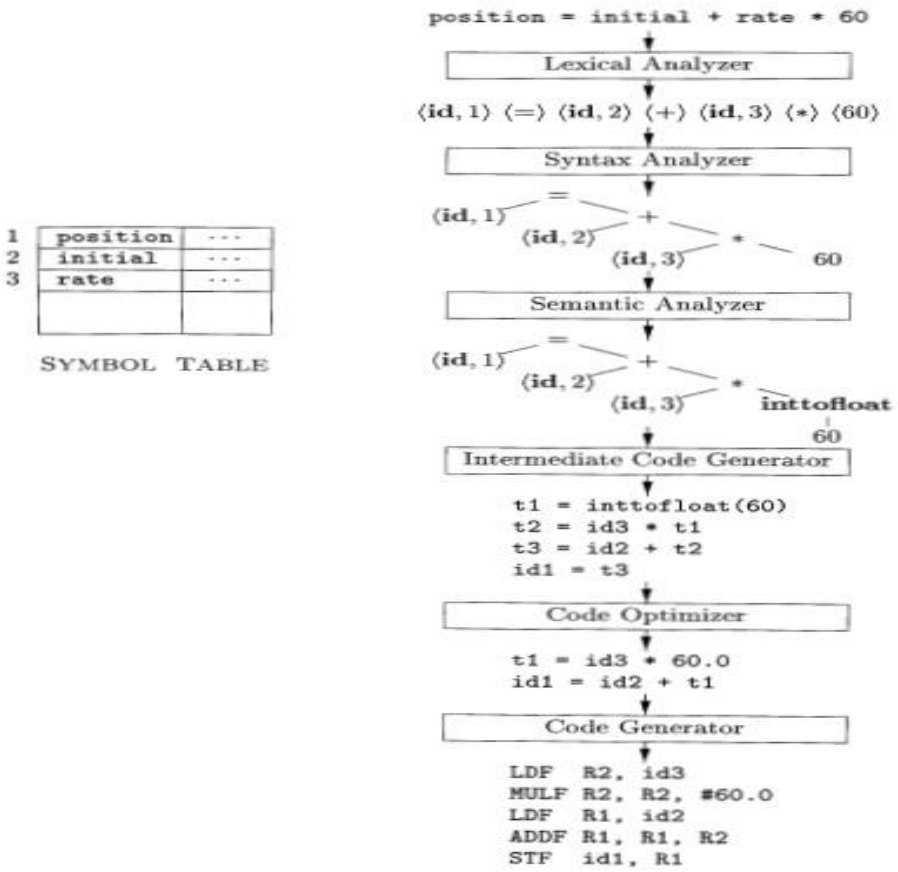


Figure 1.7: Translation of an assignment statement

b. What are the applications of Compiler? Explain. **(08 Marks)**

Solution:

1. **Implementation of High-level Programming**

A high-level programming language defines a programming abstraction: the programmer specifies an algorithm in the language, and the compiler must translate it to the target language. Higher-level programming languages are sometimes easier to develop in, but they are inefficient, therefore the target applications run slower. Low-level language programmers have more control over their computations and, in principle, can design more efficient code. Lower-level programs, on the other hand, are more difficult to build and much more difficult to maintain. They are less portable, more prone to errors, and more complex to manage.

Optimized compilers employ ways to improve the performance of generated code, compensating for the inefficiency of high-level abstractions.
In actuality, programs that utilize the register keyword may lose efficiency since programmers aren't always the best judges of extremely low-level matters like register allocation. The ideal register allocation approach is very reliant on the design of the machine. Hardwiring low-level resource management decisions like register allocation may actually harm performance, especially if the application is executed on machines that aren't meant for it.

2. **Optimization of computer architectures**
Aside from the rapid evolution of computer architectures, there is a never-ending demand for new compiler technology. Almost all high-performance computers leverage parallelism and memory hierarchies as essential methods. Parallelism may be found at two levels: at the instruction level, where many operations are performed at the same time, and at the processor level, where distinct threads of the same program are executed on different processors. Memory hierarchies address the fundamental problem of being able to produce either extremely fast storage or extremely huge storage, but not both.

*3***. Design of new computer architectures**
In the early days of computer architecture design, compilers were created after the machines were built. That isn't the case now. Because high-level programming is the norm, the performance of a computer system is determined not just by its sheer speed, but also by how well compilers can use its capabilities. Compilers are created at the processor-design stage of contemporary computer architecture development, and the resultant code is used to evaluate the proposed architectural features using simulators.

4. **Program Translations:**
The compilation is typically thought of as a translation from a high-level language to the machine level, but the same approach may be used to translate across several languages. The following are some of the most common applications of software translation technologies.

- Compiled Simulation
- Binary translation
- Hardware Syntheses
- Database Query Interpreters

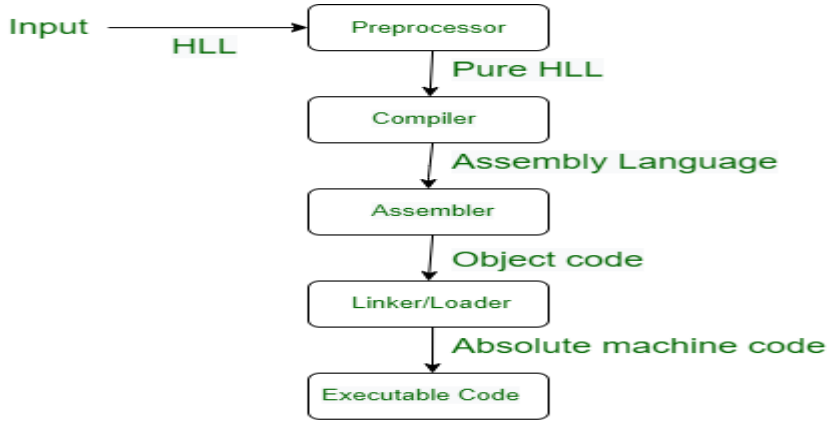
*5***. Software productivity tools**
Programs are possibly the most complex technical objects ever created; they are made up of a plethora of little elements, each of which must be accurate before the program can function properly. As a result, software mistakes are common; errors can cause a system to crash, generate incorrect results, expose a system to security threats, or even cause catastrophic failures in key systems. Testing is the most common method for discovering program flaws.

A fascinating and interesting complementary option is the use of data-flow analysis to statically discover problems (that is before the program is run). Unlike program testing, the data-flow analysis may uncover vulnerabilities along any possible execution path, not only those used by the input data sets. Many data-flow-analysis techniques, originally developed for compiler optimizations, may be used to build tools that assist programmers with their software engineering responsibilities.

4 a. Write a brief note on Language Processing System. **(06 Marks)**

Solution:

write programs in a high-level language, which is Convenient for us to comprehend and memorize. These programs are then fed into a series of devices and operating system (OS) components to obtain the desired code that can be used by the machine. This is known as a **language processing system**.



Components of Language processing system :

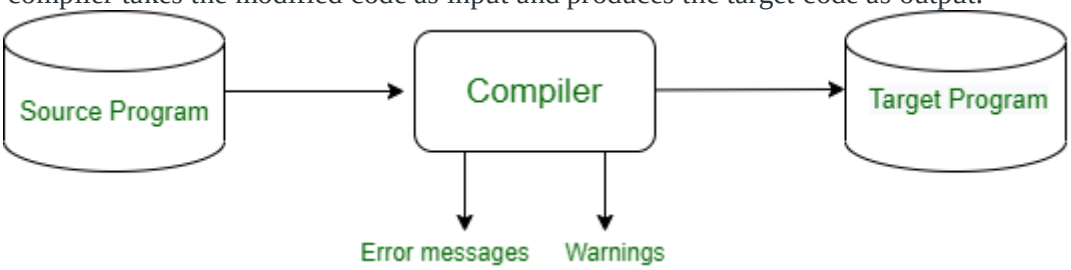
Preprocessor:–

The pre-processor includes all header files and also evaluates whether a macro(A macro is a piece of code that is given a name. Whenever the name is used, it is replaced by the contents of the macro by an interpreter or compiler.

The purpose of macros is either to automate the frequency used for sequences or to enable more powerful abstraction) is included. It takes source code as input and produces modified source code as output. The pre-processor is also known as a macro evaluator, processing is optional that is if any language that does not support #include and macros processing Is not required.

Compiler –

The compiler takes the modified code as input and produces the target code as output.



Input-Output

Assembler:

The assembler takes the target code as input and produces real locatable machine code as output.

Linker:

A linker or link editor is a program that takes a collection of objects (created by assemblers and compilers) and combines them into an executable program.

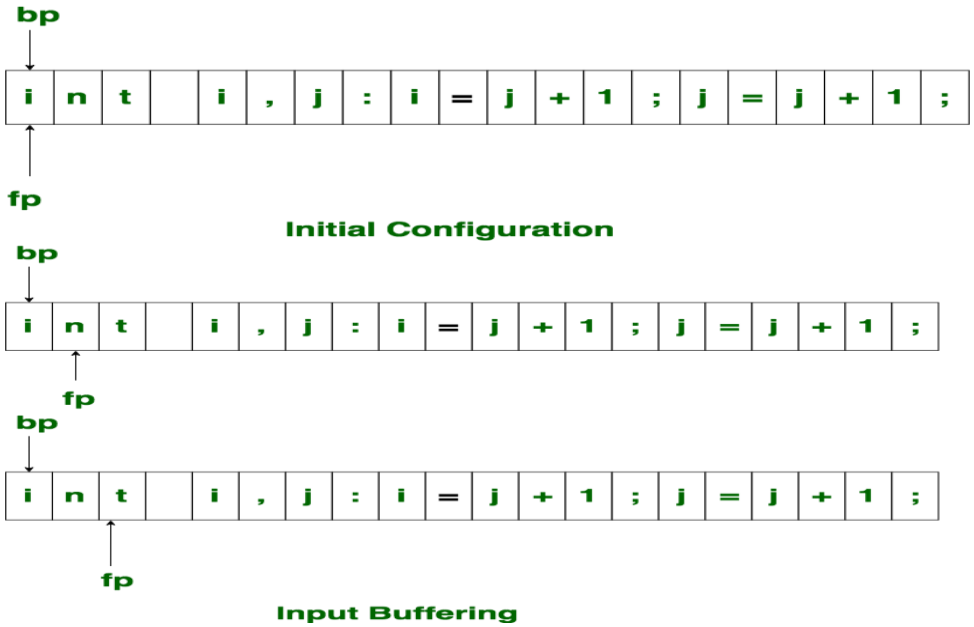
Loader:
The loader keeps the linked program in the main memory.

Executable code:

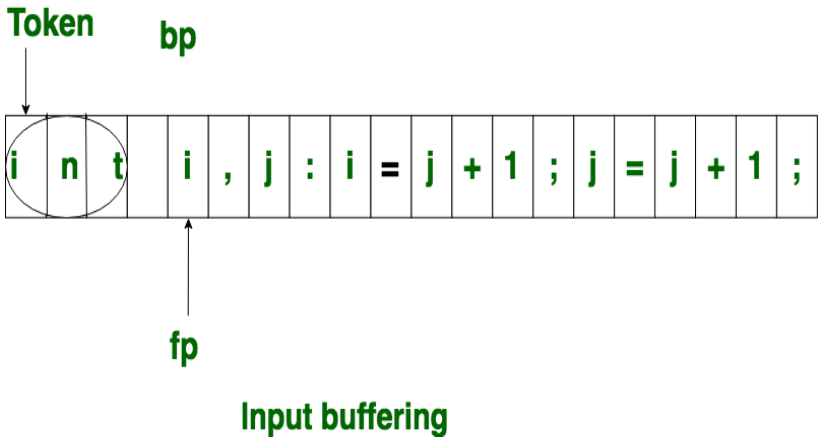
It is the low level and machine specific code and machine can easily understand. Once the job of linker and loader is done then object code finally converted it into the executable code.

b. Explain the concept of input buffering in the Lexical analysis with its implementation. (10 Marks)

Solution :
The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.

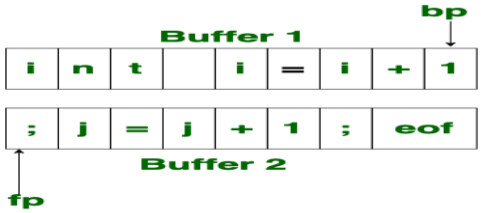


The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Input buffering

Two Buffer Scheme: To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end



Two buffer scheme storing input string

c.Define lexeme, token and pattern with example

Solution:

*A **token*** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. We will often refer to a token by its token name.

*A **pattern*** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

*A **lexeme*** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Token	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, 1,s, e	else
Comparison	comparison< or > or <= or >= or == or ! =	<=
Id	letter followed by letters and digits	Total
Numeral	any numeric constant	3.4
Literal	anything but ", surrounded by “	“hello world”

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon

Example: Consider the following C statement

printf ("Total = %d\n”, score) ;

bothprintf and score are lexemes matching the pattern for token **id,** and

"Total = %d\n” is a lexeme matching literal.

5 a. Define Context Free Grammar. Obtain CFG to generate strings of a’s and b’s having substring “ab”. **(10 Marks)**

b. Consider grammar given below from which any arithmetic expressions can be obtained.

$E \rightarrow E+E \quad E \rightarrow E-E \quad E \rightarrow E*E \quad E \rightarrow E \mid E \quad E \rightarrow id$

Show that the grammar is ambigious for the sentence id + id * id. **(10 Marks)**

Define Context free Grammar
Solution :

CFG stands for context-free grammar. It is is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

1. G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.
T is the final set of a terminal symbol. It is denoted by lower case letters.
V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).
S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

Obtain CFG to generate strings of a's and b's having a substrings ab

Solution:

S → aSb, (Rule: 1)
S → ab (Rule: 2)

First compute some strings generated by the production rules of the grammar G in the above;

(i) S ⇒ ab, (Rule: 2)

(ii) S ⇒ aSb, (Rule: 1)
⇒ aabb, (Rule: 2)
i.e. ⇒ aabb ⇒ a^2b^2

(iii)S ⇒ aSb, (Rule: 1)
⇒ aaSbb, (Rule: 1)
⇒ aaabbb, (Rule: 2)
i.e. ⇒ aaabbb ⇒ a^3b^3

(iv)S ⇒ aSb, (Rule: 1)
⇒ aaSbb, (Rule: 1)
⇒ aaaSbbb, (Rule: 1)
⇒ aaaabbbb, (Rule: 2)
i.e. ⇒ aaaabbbb ⇒ a^4b^4

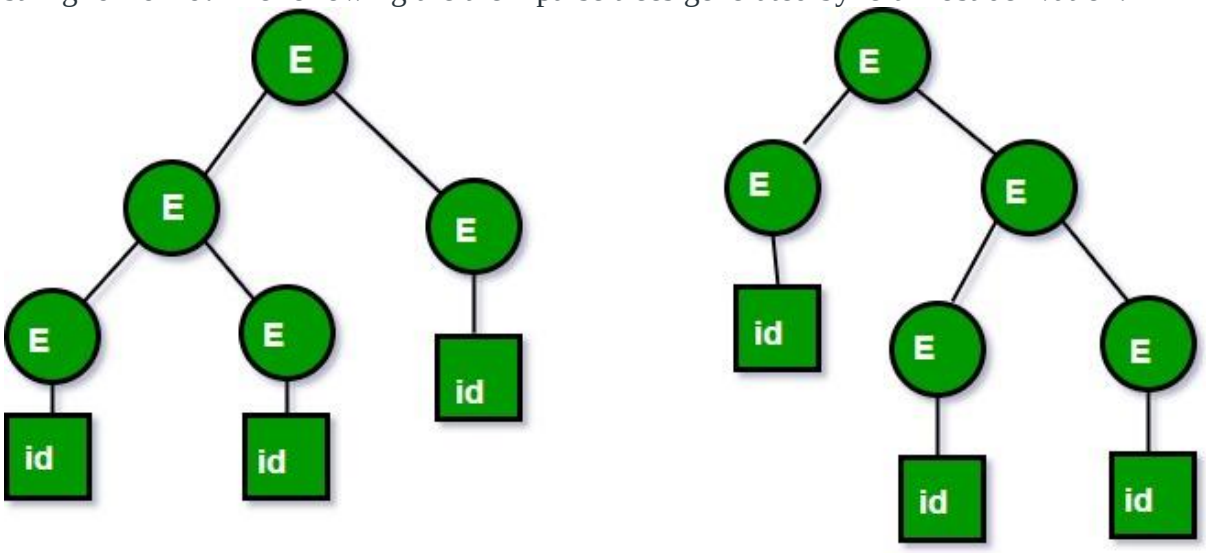
(v) S ⇒ aSb, (Rule: 1)
⇒ aaSbb, (Rule: 1)
⇒ aaaSbbb, (Rule: 1)
⇒ aaaaSbbbb, (Rule: 1)
⇒ aaaaabbbbb, (Rule: 2)
i.e. ⇒ aaabbb ⇒ a^5b^5

Hence; Language generated by the above grammar L(G) = {ab, a^2b^2, a^3b^3, a^4b^4, a^5b^5, a^6b^6, a^7b^7,.. .. .. .. }By analyzing the above generated string form the grammar G, there has a similar pattern in all computed strings, i.e.

- The length of the string is greater than or equal to 2.
- Number of a's and b's are equal.
- Presence of a's followed by b's.

Thus we can write the language of the grammar L(G) = $\{a^n b^n : n \geq 1\}$

b)
Solution:
A CFG is said to be ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT)
This grammar: **E -> E+E|id** We can create a 2 parse tree from this grammar to obtain a string **id+id+id.** The following are the 2 parse trees generated by left-most derivation:



Both the above parse trees are derived from the same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

6 a. Write an algorithm to eliminate left recursion from a grammar. Eliminate left recursion from the given grammar. $S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid \varepsilon$. **(10 Marks)**
b. Define Shift – Reduce Parser and Handle. What are conflicts in shift – reduce parse, explain with example. **(06 Marks)**
c. List and explain different actions of shift – reducer parser **(04 Marks)**

Answer:
(a)
The general form for left recursion is

$A \rightarrow A\alpha_1|A\alpha_2| \ldots . |A\alpha_m|\beta_1|\beta_2| \ldots . . \beta_n$

can be replaced by

$A \rightarrow \beta_1 A'|\beta_2 A'| \ldots . . | \ldots . . |\beta_n A'$

$A' \rightarrow \alpha_1 A'|\alpha_2 A'| \ldots . . |\alpha_m A'|\varepsilon$

Algorithm:

- Arrange non-terminals in some order: $A_1 \ldots A_n$
  - **for** i **from** 1 **to** n **do** {
  - **For j from 1 to i-1 do {**

Replace each production $A_i \rightarrow A_j \psi$
By $A_i \rightarrow \alpha_1 \psi \mid ... \mid \alpha_k \psi$
where $A_j \rightarrow \alpha_1 \mid ... \mid \alpha_k$
}

In the given question we have indirect left recursion.
Step1: Put S production in A
S → Aa | b
A→ Ac | A a d |bd | Ɛ

Step2: Eliminate left recursion from A

S → Aa | b
A→ bd A' | ƐA'
A'→ c A'| ad A'|Ɛ

(b) Shift Reduce Parser:

Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from Leaves to the Root. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.

Shift Reduce Parser requires two Data Structures

- Input Buffer
- Stack

There are the various steps of Shift Reduce Parsing which are as follows −

There are the various steps of Shift Reduce Parsing which are as follows −

- It uses a stack and an input buffer.
- Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

- **Shift** − Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
- **Reduce** − Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.
- **Accept** − Step 3 and Step 4 will be repeated until it has detected an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.

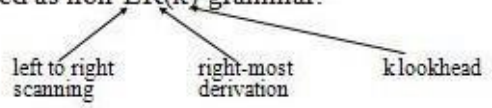
Handle:

Each replacement of the Right side of production by the left side in the process above is known as **"Reduction"** and each replacement is called **"Handle."**

Conflicts in Shift- Reduce Parser

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.



- An ambiguous grammar can never be a LR grammar.

Module-4

7 a. Explain the three basic section of LEX program with example. **(10 Marks)**
b. Write LEX program to count word, character and line count in a given file. **(10 Marks)**

Ans:

A LEX program consists of three parts:

Declarations
%
%

translation rules
%
%

auxiliary procedures

The declarations section includes declarations of variables,constants,and regular definitions.

The translation rules of a lex program are statements of the form

R1 {action 1}
R2 {action 2}
.... ……
….
Rn {action n} where each Ri is regular expression and each action i, is a program fragment describing what action the lexical analyzer should take when pattern Ri matches lexeme. Typically, action i will return control to the parser. In Lex actions are written in C;in general,however,they can be in any implementation language.

The third section holds whatever auxiliary procedures are needed by the actions.

(b)

```
/*Lex Program to count numbers of lines, words, spaces and
characters



in a given statement*/

%{

#include<stdio.h>

int sc=0,wc=0,lc=0,cc=0;

%}



%%

[\n] { lc++; cc+=yyleng;}

[  \t] { sc++; cc+=yyleng;}

[^\t\n ]+ { wc++;  cc+=yyleng;}

%%



int main(int argc ,char* argv[ ])
```

```
{
        printf("Enter the input:\n");
        yylex();
        printf("The number of lines=%d\n",lc);
        printf("The number of spaces=%d\n",sc);
        printf("The number of words=%d\n",wc);
        printf("The number of characters are=%d\n",cc);
}


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

8 a. What is YACC? Explain the different sections used in writing the YACC specification. Explain with example program. **(10 Marks)**

b. Define Regular Expression. What is the use of following Meta characters :
i) · ii) * iii) ∧ iv) $ v) { } vi) ? **(07 Marks)**

c. Discuss how Lexes and Parser communicate. **(03 Marks)**

Ans:

The UNIX utility yacc (Yet Another Compiler Compiler) parses a stream of token, typically generated by lex, according to a user-specified grammar.

Structure of a YACC file

A yacc file looks much like a lex file:

definitions

%%
rules
%%
code

Definition: All code between %{ and %} is copied to the beginning of the resulting C file.

Rules: A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.

Code: This can be very elaborate, but the main ingredient is the call to **yylex**, the lexical analyzer. If the code segment is left out, a default main is used which only calls **yylex.**

Definition section

There are three things that can go in the definitions section:

C code: Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions: The definition section of a lex file was concerned with characters; in **yacc** this is tokens.
Example: **%token NUMBER.**
These token definitions are written to a **.h** file when yacc compiles this file.

Associativity rules These handles associativity and priority of operators.

Lex Yacc interaction

Conceptually, **lex** parses a file of characters and outputs a stream of tokens; **yacc** accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.

If your lex program is supplying a tokenizer, the yacc program will repeatedly call the yylex routine. The lex rules will probably function by calling return every time they have parsed a token.

If **lex** is to return tokens that **yacc** will process, they have to agree on what tokens there are. This is done as follows:

For Example

1.The yacc file will have token definition **%token NUMBER** in the definitions section.

2. When the yacc file is translated with **yacc –d** , a header file **y.tab.h** is created that has definitions like **#define NUMBER 258.**

3. The **lex** file can then call return **NUMBER**, and the **yacc** program can match on this token.

Rules section

The rules section contains the grammar of the language you want to parse. This looks like

```
statement : INTEGER '=' expression
          | expression
          ;
expression : NUMBER '+' NUMBER
           |        NUMBER '-' NUMBER
           ;
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically defines coming from **%token** definitions in the yacc program or character values.

(b)

.	Matches any character except \n.
*****	Match zero or more occurrences of the preceding pattern.
?	Matches zero or one occurrences of the preceding pattern.
	Ex: -?[0-9]+ matches a signed number including an optional leading minus.
{ }	1) Indicates how many times a pattern can be present. Example: A{1,3} implies
	one to three occurrences of A may be present.
	2) If they contain name, they refer to a substitution by that name.
	Ex: {digit}
$	Matches end of line as the last character of the pattern.
^	Negation.

Module-5

9 a. Define S – Attribute and I – Attribute with respect to SDD and construct Syntax tree, Parse tree and annotated tree for string 5 *6 + 7 by using given grammar.

$S \rightarrow En$
$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F$
$T \rightarrow T \mid F$
$T \rightarrow F$
$F \rightarrow (E) \mid digit \mid$
$n \rightarrow ;$

(10 Marks)

b. What are the different three address code instructions? Translate the arithmetic expression a + b – (-c) into quadruples , triplets and indirect triples. **(10 Marks)**

(a)

S-Attributed Definitions.

An SDD is S-attributed if every attribute is synthesized.
If an SDD is S-attributed, we evaluate its attributes in any bottom-up ordering of the parse tree nodes.
It is simpler to perform a post-order tree traversal and evaluate the attributes at a node *N* when the traversal leaves *N* for the last time.
These definitions are implemented during bottom-up parsing because a bottom-up parse corresponds to a post-order traversal, in other words, a post order traversal corresponds to the order that an LR parser reduces the production body to its head.

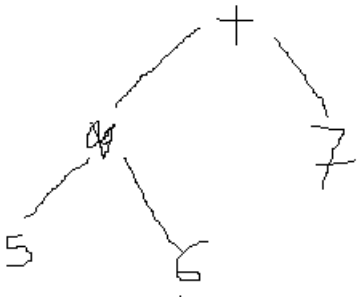
L-Attributed Definitions.

The idea is that between attributes associated with a production body, the edges of a dependency graph can go from right to left but not the other way round(left to right), hence the name 'L-attributed'.

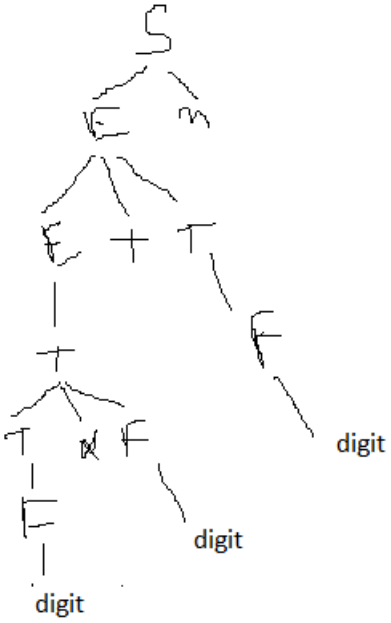
In other words, each attribute must either be;

1. Synthesized, or,
2. Inherited but with limited rules, i.e Suppose there is a production $A \rightarrow X1X2...Xn$ and an inherited attribute $Xi.a$ computed by a rule associated with this production, then the rule only uses;
   ** inherited attributes that are associated with head *A*.
   ** Either inherited attribute or synthesized attributes associated with the occurrences of symbols *X1,X2, ..., Xi-1* located to the left of *Xi*.
   ** Inherited or synthesized attributes that are associated with such an occurrence of *Xi* itself, only in such a way that no cycles exist in the dependency graph formed by *Xi* attributes.

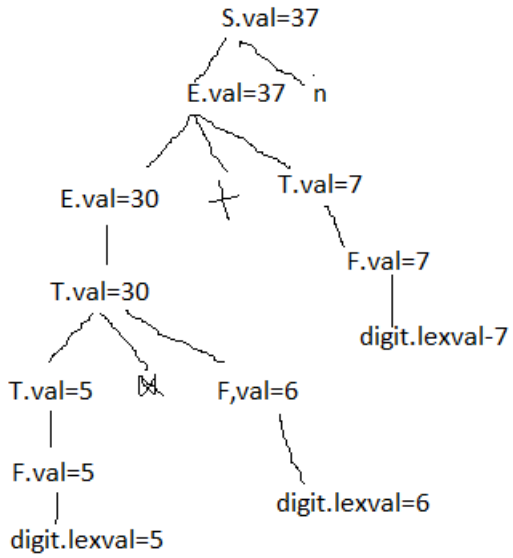
Syntax Tree



Parse Tree



Annotated Parse Tree



(b)

Common Three Address Instruction Forms-

The common forms of Three Address instructions are-

1. Assignment Statement-

x = y op z and x = op y

Here,

- x, y and z are the operands.
- op represents the operator.

It assigns the result obtained after solving the right-side expression of the assignment operator to the left side operand.

2. Copy Statement-

x = y

Here,

- x and y are the operands.
- = is an assignment operator.

It copies and assigns the value of operand y to operand x.

3. Conditional Jump-

If x relop y goto X

Here,

- x & y are the operands.
- X is the tag or label of the target statement.
- relop is a relational operator.

If the condition “x relop y” gets satisfied, then-

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

If the condition “x relop y” fails, then-

- The control is not sent to the location specified by label X.
- The next statement appearing in the usual sequence is executed.

4. Unconditional Jump-

goto X

Here, X is the tag or label of the target statement.

On executing the statement,

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

5. Procedure Call-

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

Quadruples

Operator	Arg1	Arg2	Result
Uminus	c		T1
+	a	b	T2
-	T2	T1	T3

Triplets

Operator	Arg1	Arg2
Uminus	c	
+	a	b
-	(1)	(0)

Indirect Triples

100	**(0)**
101	**(1)**
102	**(2)**

10 a. Define SDD. Give SDD for simple type declaration. Construct a dependency graph for the declaration int a, b ; **(10 Marks)**

b. Explain the issues in design of code generation. **(10 Marks)**

Solution:

Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions. It is a context free grammar with attributes and rules together which are associated with grammar symbols and productions respectively.

The process of syntax directed translation is two-fold:

Construction of syntax tree and

• Computing values of attributes at each node by visiting the nodes of syntax tree.

Semantic actions

Semantic actions are fragments of code which are embedded within production bodies by syntax directed translation.

They are usually enclosed within curly braces ({ }).

It can occur anywhere in a production but usually at the end of production.

(eg.)

$$E \longrightarrow E_1 + T \{print\ '+'\}$$

Types of translation

• L-attributed translation

o It performs translation during parsing itself.

o No need of explicit tree construction.

o L represents 'left to right'.

• S-attributed translation

o It is performed in connection with bottom up parsing.

o 'S' represents synthesized.

Production	Semantic Rules

D —>TL	*L.inh* = *T.type*
T —> *int*	*T.type* =integer
T —> *float*	*T.type* = *float*
L —> L_1, **id**	L_1.inh = *L.inh* *addType (id.entry, Linh)*
L —> **id**	*addType (id.entry, L.inh)*

(b) Issues in Code Generation

In the code generation phase, various issues can arises:

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

Input to the code generator

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

1. MOV b, R0 R0→b
2. ADD c, R0 R0 c + R0
3. MOV R0, a a → R0
4. MOV a, R0 R0→ a

5. ADD e, R0 R0 → e + R0
6. MOV R0, d d → R0

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.