USN | | | | | | | | | |

Solution to Internal Assessment Test III – May. 2019

Sub:	System Software & Compiler Design					Sub Code:	15CS63	Branch:	CSE	
Date:	14/05/2019	Duration:	90 min's	Max Marks:	50	Sem/Sec:	6/CSE(A,B,C)			OBE

1.a) What is loader? What are advantages and disadvantages? Explain boot strap loader with algorithm.

Solution:

In computer systems a **loader** is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution.

Advantages

1. When source program is executed an object program getsgenerated. So there is no need to retranslate the program each time.

2. Instead of placing the assembler in the memory the loaderoccupies a portion of the memory. A loader is smaller than assembler so thereis no wastage of memory.

3. The source program can be written with multiple programsand multiple languages.

Disadvantages

1. If the program is modified it has to be retranslated.

2. Some portion of the memory is occupied by the loader.

A simple SIC/XE bootstrap loader

1. The bootstrap itself begins at address 0 in the memory of the machine
2. It loads the OS (or some other program) starting address 0x80
3. The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.

4. After all the object code from device F1 has been loaded, the bootstraps jumps to address 80
5. Begin the execution of the program that was loaded.
6. begin
7. X=0x80 ; the address of the next memory location to be loaded
8. Loop
9. A←GETC ; read one char. From device F1 and convert it from the
10. ; ASCII character code to the value of the hex digit
11. Save the value in the high-order 4 bits of S
12. A←GETC
13. A←(A+S) ; combine the value to form one byte
14. store the value (in A) to the address represented in register X
15. X←X+1
16. End

1.b) Enlist any four different loader option commands

Solution:

1. INCLUDE program-name(library-name)

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.

2. DELETE csdect-name

Instruct the loader to delete the named control section(s) from the set of programs being loaded.

3. CHANGE name1, name2

Cause the external symbol *name1* to be changed to *name2* wherever it appears in the object programs

INCLUDE READ(UTLIB)

INCLUDE WRITE(UTLIB)

DELETE RDREC, WRREC

CHANGE RDREC, READ

CHANGE WRREC, WRITE

17. 4.LIBRARY MYLIB

Automatic inclusion of library routines to satisfy external references Searched before the standard libraries

NOCALL STDDEV, PLOT, CORREL

To instruct the loader that these external references are to remain unsolved.

2. Define and Explain the following:

i) Linking loader **ii) Dynamic linking**

Solution:

i) Linking loader

A linking loader usually makes two passes

- Pass 1 assigns addresses to all external symbols by creating ESTAB.
- Pass 2 performs the actual loading, relocation, and linking by using ESTAB.
- The main data structure is ESTAB (hashing table).

A linking loader usually makes two passes

- ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded.
- Two variables PROGADDR and CSADDR.
- PROGADDR is the beginning address in memory where the linked program is to be loaded.
- CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

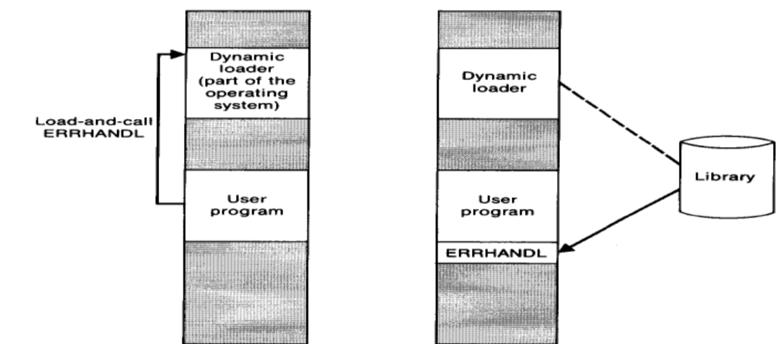
The linking loader algorithm,

- In Pass 1, concerned only Header and Defined records.
- ➢ CSADDR+CSLTH = the next CSADDR.
- ➢ A load map is generated.
- ➢ In Pass 2, as each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

- When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.
  - This value is then added to or subtracted from the indicated location in memory.

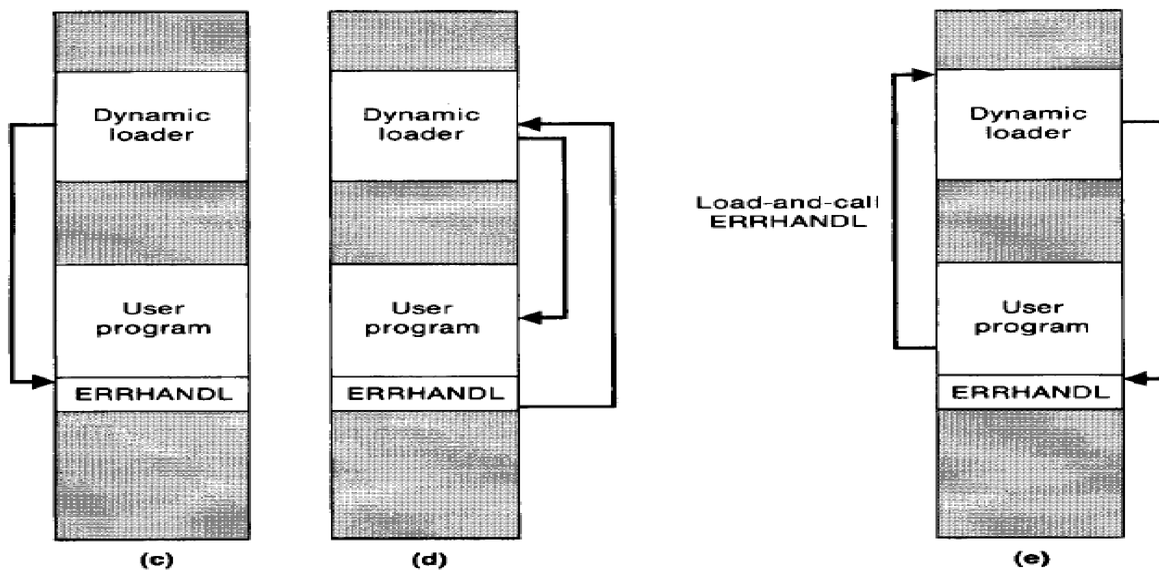
Dynamic linking (dynamic loading, load on call)

- Postpones the linking function until execution time.
- A subroutine is loaded and linked to the rest the program when is first loaded.
- Dynamic linking is often used to allow several executing program to share one copy of a subroutine or library.
  1. Run-time library (C language), dynamic link library
  2. A single copy of the routines in this library could be loaded into the memory of the computer.
- Dynamic linking provides the ability to load the routines only when (and if) they are needed.
  1. For example, that a program contains subroutines that correct or clearly diagnose error in the input data during execution.
  2. If such error are rare, the correction and diagnostic routines may not be used at all during most execution of the program.
  3. However, if the program were completely linked before execution, these subroutines need to be loaded and linked every time.
- Dynamic linking avoids the necessity of loading the entire library for each execution.
- Following Fig. illustrates a method in which routines that are to be dynamically loaded must be called via an operating system (OS) service request.



- The program makes a load-and-call service request to OS.

- The parameter argument (ERRHANDL) of this request is the symbolic name of the routine to be loaded.
- OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded form the specified user or system libraries.
- Control id then passed form OS to the routine being called.
- When the called subroutine completes its processing, OS then returns control to the program that issued the request.
- If a subroutine is still in memory, a second call to it may not require another load operation.



3. Write SIC/XE source code for bootstrap loader and explain it.

Solution:

Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called a bootstrap loader is executed

In PC, BIOS acts as a bootstrap loader This bootstrap loads the first program to be run by the computer -- usually an operating system

A simple SIC/XE bootstrap loader

- The bootstrap itself begins at address 0 in the memory of the machine
    - It loads the OS (or some other program) starting address 0x80
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
- After all the object code from device F1 has been loaded, the bootstraps jumps to address 80
- Begin the execution of the program that was loaded.

```
BOOT    START   0       BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
        CLEAR   A       CLEAR REGISTER A TO ZERO
        LDX    #128     INITIALIZE REGISTER X TO HEX 80
LOOP    JSUB    GETC    READ HEX DIGIT FROM PROGRAM BEING LOADED
        RMO     A,S     SAVE IN REGISTER S
        SHIFTL  S,4     MOVE TO HIGH-ORDER 4 BITS OF BYTE
        JSUB    GETC    GET NEXT HEX DIGIT
        ADDR    S,A     COMBINE DIGITS TO FORM ONE BYTE
        STCH    0,X     STORE AT ADDRESS IN REGISTER X
        TIXR    X,X     ADD 1 TO MEMORY ADDRESS BEING LOADED
        J       LOOP    LOOP UNTIL END OF INPUT IS REACHED
```

```
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC     TD      INPUT     TEST INPUT DEVICE
         JEQ     GETC      LOOP UNTIL READY
         RD      INPUT     READ CHARACTER
         COMP    #4        IF CHARACTER IS HEX 04 (END OF FILE),
         JEQ     80            JUMP TO START OF PROGRAM JUST LOADED
         COMP    #48       COMPARE TO HEX 30 (CHARACTER '0')
         JLT     GETC      SKIP CHARACTERS LESS THAN '0'
         SUB     #48       SUBTRACT HEX 30 FROM ASCII CODE
         COMP    #10       IF RESULT IS LESS THAN 10, CONVERSION IS
         JLT     RETURN       COMPLETE. OTHERWISE, SUBTRACT 7 MORE
         SUB     #7           (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN   RSUB              RETURN TO CALLER
INPUT    BYTE    X'F1'     CODE FOR INPUT DEVICE
         END     LOOP
```

Figure 3.3 Bootstrap loader for SIC/XE.

```
begin
    X=0x80         ; the address of the next memory location to be loaded
  Loop
    A←GETC         ; read one char. From device F1 and convert it from the
                   ; ASCII character code to the value of the hex digit
    save the value in the high-order 4 bits of S
    A←GETC
    A← (A+S)       ; combine the value to form one byte
    store the value (in A) to the address represented in register X
    X←X+1
end
```

4. Define SDD and SDT. Write SDD for simple desk calculator and show annotated parser tree for the expression (3+4) * (5+6)

Solution:

Syntax-Directed Definitions A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

Example: PRODUCTION SEMANTIC RULE E → E1 + T E.code = E1.code || T.code || '+'

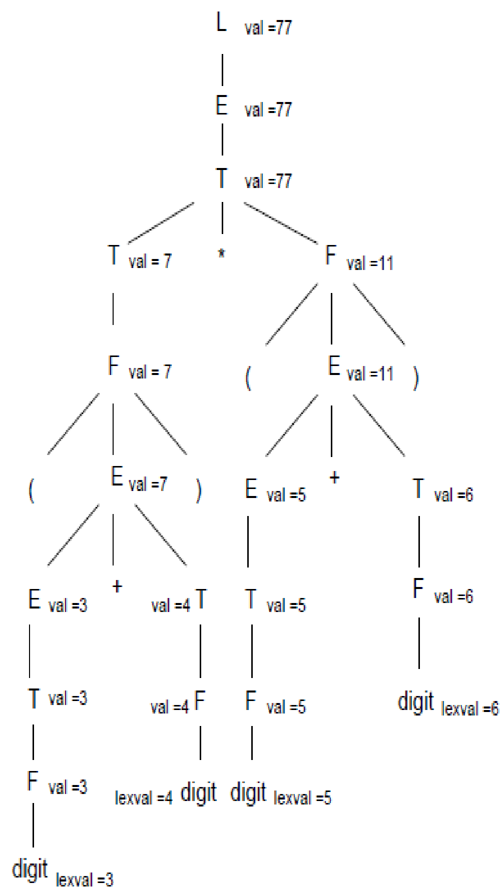
Syntax-Directed Translation Schemes (SDT) SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed

Example: In the rule E → E1 + T { print '+' }, the action is positioned after the body of the production.

Write SDD for simple desk calculator and show annotated parser tree for the expression (3+4) * (5+6)

Solution:

$L \rightarrow E$	$L.val = E.val$
$E \rightarrow T$	$E.val = T.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$T \rightarrow F$	$T.val = F.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$



6. **Generate Intermediate Code for the following statements and identify the basic [10] blocks .given w=8 bytes**

for i from 1 to 10 do

for j from 1 to 10 do

a[i,j]=0.0

for i from 1 to 10 do.

a[i,i]=1.0

Solution:

```
1)  i = 1
     2)  j = 1
     3)  t1 = 10 * i
     4)  t2 = t1 + j        // element [i,j]
     5)  t3 = 8 * t2        // offset for a[i,j] (8 byte reals)
     6)  t4 = t3 - 88       // program array starts at [1,1] assembler at [0,0]
     7)  a[t4] = 0.0
     8)  j = j + 1
     9)  if j <= 10 goto (3)
    10)  i = i + 1
    11)  if i <= 10 goto (2)
    12)  i = 1
    13)  t5 = i - 1
    14)  t6 = 88 * t5
    15)  a[t6] = 1.0
    16)  i = i + 1
    17)  if i <= 10 goto (13)
```

Which quads are leaders?

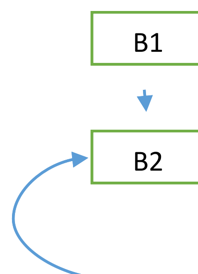
1 is a leader by definition. The jumps are 9, 11, and 17. So 10 and 12 are leaders as are the targets 3, 2, and 13.

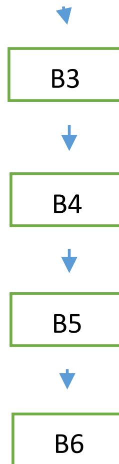
The leaders are then 1, 2, 3, 10, 12, and 13.

The basic blocks are therefore {1}, {2}, {3,4,5,6,7,8,9}, {10,11}, {12}, and {13,14,15,16,17}.

Here is the code written again with the basic blocks indicated.

```
1)   i = 1-B1

2)   j = 1-B2

3)   t1 = 10 * i
4)   t2 = t1 + j  -B3          // element [i,j]
5)   t3 = 8 * t2             // offset for a[i,j] (8 byte numbers)
6)   t4 = t3 - 88            // we start at [1,1] not [0,0]
7)   a[t4] = 0.0
8)   j = j + 1
9)   if J <= 10 goto (3)

10)  i = i + 1
11)  if i <= 10 goto (2) -B4

12)  i = 1-B5

13)  t5 = i - 1
14)  t6 = 88 * t5-B6
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```





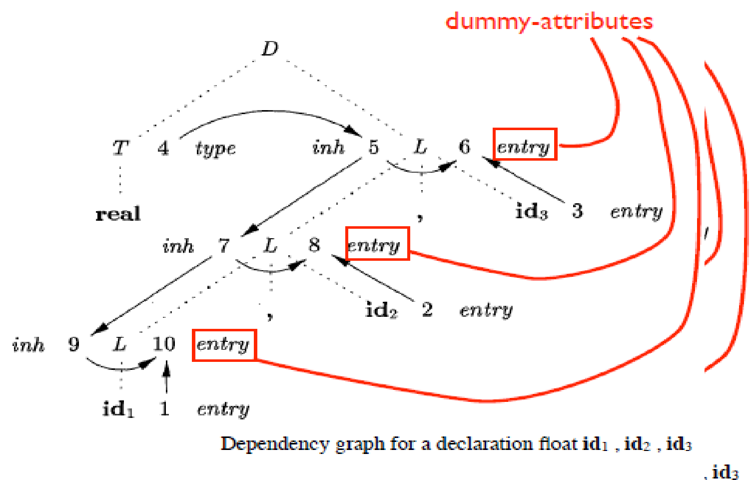
5.a)**Construct dependency graph for declaration float id1,id2,id3**

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType(\textbf{id}.entry, L.inh)$

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. *id.entry*, a lexical value that points to a symbol-table object, and
2. *L. inh*, the type being assigned to every identifier on the list.

We suppose that function *addType* properly installs the type *L.inh* as the type of the represented identifier.



Dependency graph for a declaration float id_1 , id_2 , id_3

, id_3

5.b) Define 1) Synthesis attribute 2) Inherited attribute with example

Each grammar symbol is associated with a set of attributes

computed w.r.t. the parsing tree

<**A,N**>a non terminal **A** labelling a node **N** of the parse tree

‣ **Synthesized attribute of**<**A,N**>: defined in terms of the attributes ofthe **children** of **N** and of **N itself** (semantic rule associated to the production relative to **N**)

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit.lexval**

SSD for a desk calculator

In this case each non terminalsymbol has a**unique synthesizedattribute***val*

‣ **Inherited attribute of**<**A,N**>: defined in terms of the **N's parent**, **Nitself**, and **N's siblings** (semantic rule associated to the productionrelative to the parent of **N**)

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow digit$	$F.val = digit.lexval$

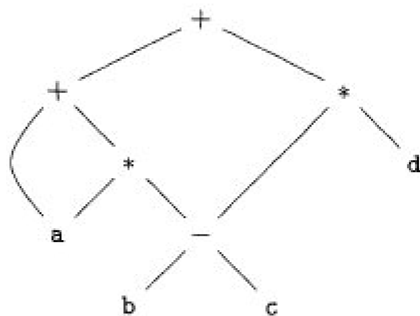
Here both inherited and synthesis attribute T.inh – values obtained from sibling from left to right

T.val-synthesis attribute –value assign from child to node N

- **General attribute**: value can be depended on the attributes of any nodes.

Terminal symbols can have synthetized attributed (computed by the lexical analyzer) but not inherited attributes.

7.a)Obtain DAG for the expression a+a *(b-c) +(b-c)*d



7.b) Explain Syntax Directed Translation of switch statement

Solution

```
switch ( E ) {
        case V1: S1
        case V2: S2
            ...
        case Vn-1: Sn-1
        default: Sn
}
```

```
         code to evaluate E into t
         goto test
L1:      code for S1
         goto next
L2:      code for S2
         goto next
         ...
Ln-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:
```

Figure 6.49: Translation of a switch-statement

To translate into the form of Fig. 6.49, when we see the keyword **`switch`**, we generate two new labels **`test`** and **`next`**, and a new temporary *t.* Then, as we parse the expression *E,* we generate code to evaluate *E* into *t.* After processing *E,* we generate the jump **`goto test`**.

Then, as we see each **`case`** keyword, we create a new label *Li* and enter it into the symbol table. We place in a queue, used only to store cases, a value-label pair consisting of the value *Vi* of the case constant and *Li* (or a pointer to the symbol-table entry for *Li).* We process each statement **`case`** *Vi: Si* by emitting the label *Li*attached to the code for *Si,* followed by the jump **`goto next`**

When the end of the switch is found, we are ready to generate the code for the n-way branch. Reading the queue of value-label pairs, we can generate a sequence of three-address statements of the form shown in Fig. 6.51. There, *t* is the temporary holding the value of the selector expression *E,* and h_n is the label for the default statement

.

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
label next
```

Figure 6.51: Case three-address-code instructions used to translate a switch-statement

The case t *Vi Li* instruction is a synonym for if t = *Vi* goto *Li* in Fig. 6.49, but the case instruction is easier for the final code generator to detect as a candidate for special treatment. At the code-generation phase, these sequences of case statements can be translated into an n-way branch of the most efficient type, depending on how many there are and whether the values fall into a small range.

8.a)Explain the following with a example 1) quadruple 2)triple 3)indirect triple 4) single assignment form

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Example – Consider expression a = b * – c + b * – c.
The three address code is:

```
t1 = uminus c

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5
```

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t5
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –
This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.
Example – Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

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

Indirect Triples representation

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
Two distinctive aspects that distinguish SSA from three-address code.
a. All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

Figure shows the same intermediate program in three-address code and instatic single-assignment form.

Three-address code	Static single-assignment form
p=a+b	p1=a+b
q=p-c	q1=p1-c
p=q*d	p2=q1 *d
p=e-p	p3=e-p2
q=p+q	q2=p3 +q1

8.b) List and explain different common three address form with example.

Solution:

list of the common three-address instruction forms:

1. Assignment instructions of the form *x = y op z,* where *op* is a binary arithmetic or logical operation, and *x, y,* and z are addresses.
2. Assignments of the form *x = op y,* where *op* is a unary operation. Examples of unary operations are unary minus, logical negation, shift operators, and conversion operators.
3. *Copy instructions* of the form *x = y,* where *x* is assigned the value of *y.*
4. An unconditional jump goto*L.* The three-address instruction with label *L* is the next to be executed.

5. Conditional jumps of the form if*x*goto*L*and if False *x* goto*L.*
6. These instructions execute the instruction with label L if *x* is true and false, respectively. The following three-address instruction in sequence is executed next
7. Conditional jumps such as if *x relop*ygoto*L,*

which apply a relational operator (<, ==, >=, etc.) to *x* and y, and execute the instruction with label *L* if the relation is satisfied. If not three-address instruction following if *x relop*ygoto*L*is executed next, in sequence.

8. Procedure calls and returns are implemented using the following instructions: param*x*for parameters; call p, n and y = call p, n for procedure and function calls, respectively; and return y, where y, representing a returned value, is optional. Their typical use is as the sequence of three- address instructions

param x_1

param X2

• • •

paramXn

call *p,n*

9. Generated as part of a call of the procedure $p(x_i,x_2,\ldots,x_n)$. The integer n, indicating the number of actual parameters in "call *p, n,"* is not redundant because calls can be nested.
10. Indexed copy instructions of the form *x* = y [i] and *x [i] = y.* The instruction *x*= y *[i]* sets *x* to the value in the location i memory units beyond location y. The instruction *x* [i] =y sets the contents of the location iunits beyond *x* to the value of y.
11. Address and pointer assignments of the form *x =&y* *x = *y,* and * *x* = y.

Example 6.5 : Consider the statement

do i = i+1; while (a[i] < v);

Two possible translations of this statement are shown in figure below

L: t1=i+1	100: t1=i+1
i=t1	101: i=t1
t2= i*8	102 : t2= i*8
t3 =a[t2]	103 : t3 =a[t2]

if t3 < V goto L	104: if t3 < V goto 100

Figure Two ways of assigning labels to three-address codes