CSL302: Compiler Design

End Sem Examination Solutions (2025-26-M Semester)

Max. Points: 100 **Duration:** 3 hours

November 28, 2025

1. All questions are compulsory
2. Make your assumptions and state them wherever necessary

Question-1 (Semantic Analysis): **[15 Points]**

Part-(a) //7 Marks

Consider the following grammar that generates expressions for a number of operators. Assume you have synthesized attributes **E.min** and **E.max** which should be set to the minimum and maximum values for E, and synthesized attribute **CONST.val** which is the value of the constant. Write the semantic rules that calculate the range (minimum and maximum values) of each subexpression in the respective attributes.

E	→	CONST	{ E.min = ?? ; E.max = ?? }
	|	ID	{ E.min = ID.min ; E.max = ID.max }
	|	E1 + E2	{ E.min = ?? ; E.max = ?? }
	|	E1 − E2	{ E.min = ?? ; E.max = ?? }
	|	E1 ∗ E2	{ E.min = ?? ; E.max = ?? }
	|	−E1	{ E.min = ?? ; E.max = ?? }

Solution:

E → CONST //0.5 Marks
{ E.min = CONST.val ;
E.max = CONST.val }

E → ID
{ E.min = ID.min ;
E.max = ID.max }

E → E1 + E2 //1 Mark
{ E.min = E1.min + E2.min ;
E.max = E1.max + E2.max }

E → E1 - E2 //1 Mark
{ E.min = E1.min - E2.max ; /* smallest = smallest minus largest */

```
     E.max = E1.max - E2.min }   /* largest  = largest  minus smallest */

E → E1 * E2     //4 Marks
   {
     p1 = E1.min * E2.min ;
     p2 = E1.min * E2.max ;
     p3 = E1.max * E2.min ;
     p4 = E1.max * E2.max ;
     E.min = min(p1, p2, p3, p4) ;
     E.max = max(p1, p2, p3, p4)
   }

E → - E1  //0.5 Marks
   { E.min = - E1.max ;
     E.max = - E1.min }
```

Part-(b): 8 Marks.

Let synthesized attribute F.val give the value of the binary fraction generated by F in the grammar that follows:

F →   .L
L →   LB | B
B →   0 | 1

For instance, on input .101 we have that F.val = .625

(a) Using only synthesized attributes, write the semantic rules to compute F.val for the above grammar.
(b) Show the translation of the input string .101 by annotating its parse tree

Solution:

(a) Semantic rules: //5 Marks

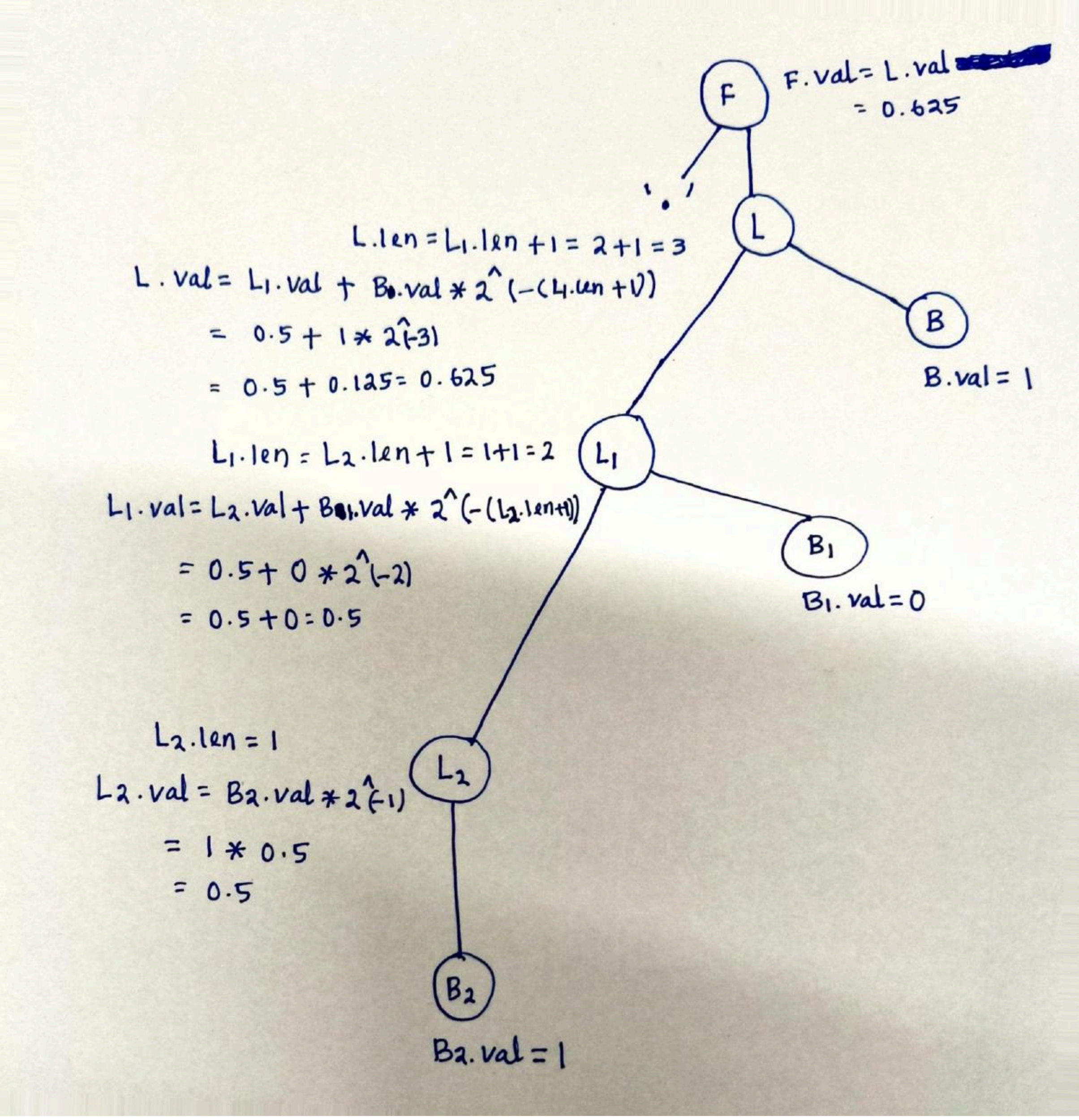
```
B → '0'    { B.val = 0 }  //0.5 Marks
B → '1'    { B.val = 1}

L → B      { L.val = B.val * 2^(-1);  //1 Mark
               L.len = 1 }

L → L1 B   { L.val = L1.val + B.val * 2^(-(L1.len + 1));  //3 Marks
               L.len = L1.len + 1 }

F → '.' L  { F.val = L.val } //0.5 Marks
```

(b) Annotated Parse Tree for .101 //3 Marks (Each level 1 Mark)



Question-2 (Syntax Analysis): **[20 Points]**

Consider the following grammar.

S → B
B → B E +
| −E
E → e
| Ee
| ∈

(a) Is the grammar LR(1)? Justify your answer by showing DFA and the LR(1) parser table.

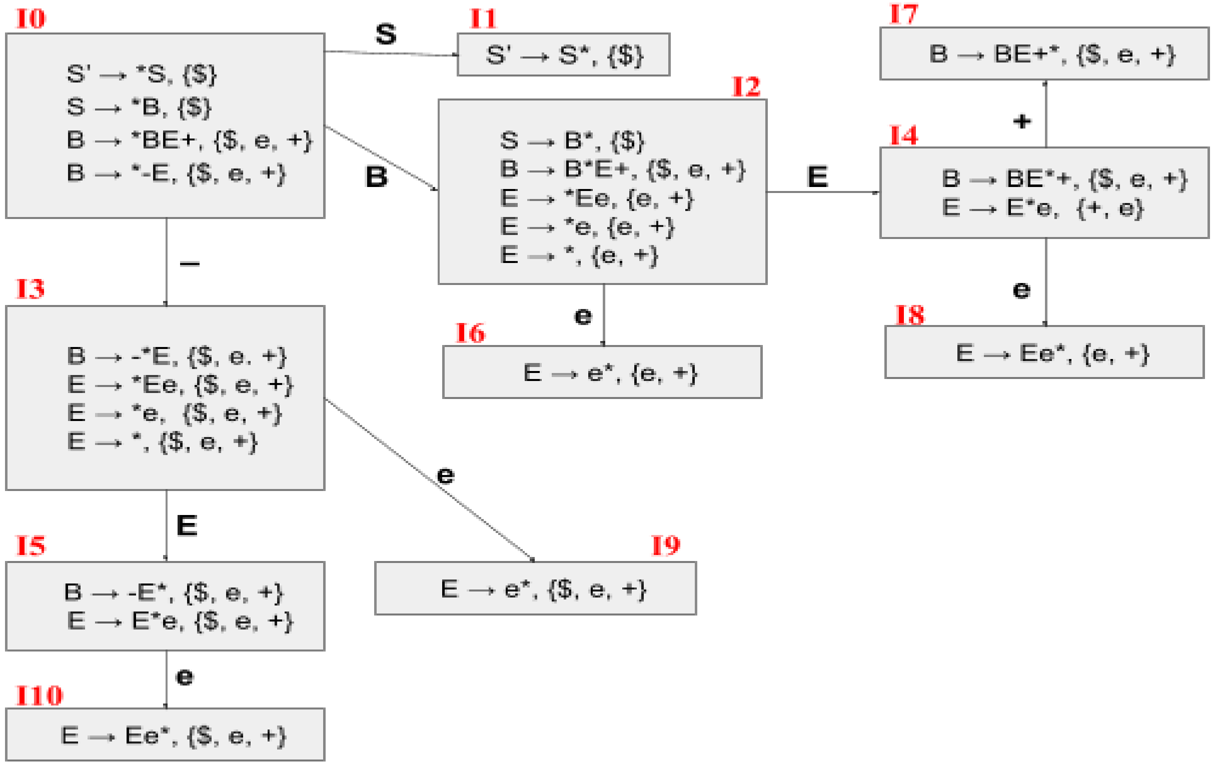
(b) Is the grammar LALR? Justify your answer by reconstructing DFA and the corresponding LALR parser table.

Solution:

Part (a):

(1) S' → S (*Augmented*)
(2) S → B
(3) B → BE+
(4) B → – E
(5) E → e
(6) E → Ee
(7) E → ε

[10 Marks]



LR(1) parser table:

[5 Marks]

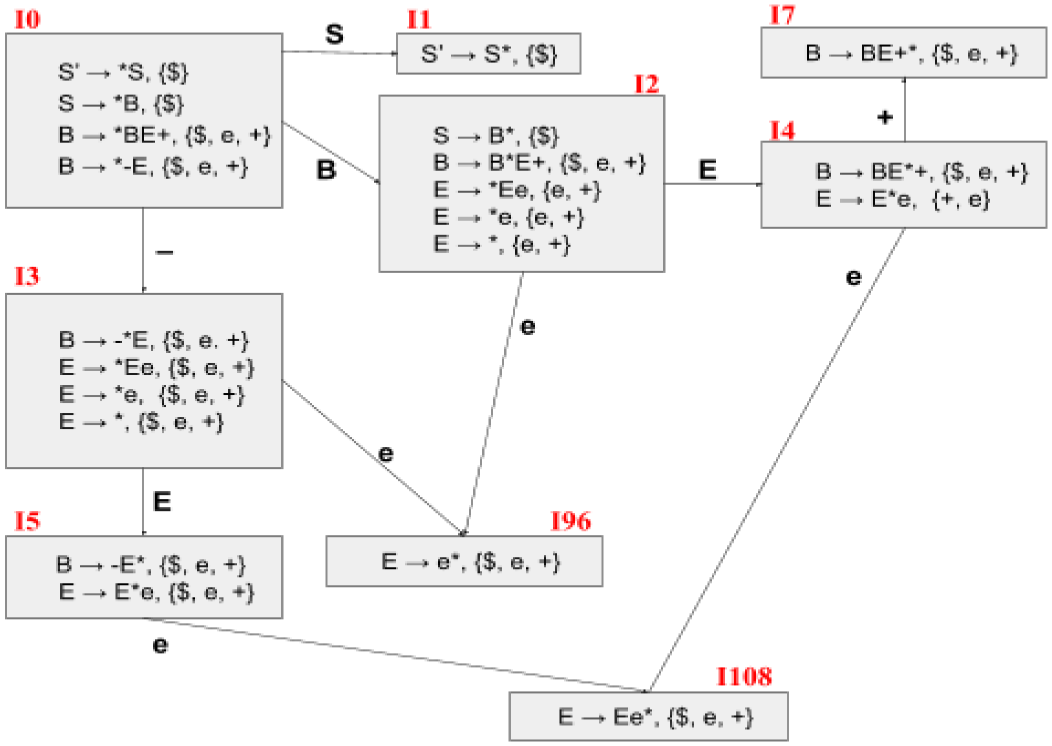
	–	+	e	$	S	B	E
I0	s3				1	2	
I1				Accept			
I2		r7	s6, r7	r7			4
I3		r7	s9, r7	r7			5
I4		s7	s8				
I5		r4	s10, r4	r4			
I6		r5	r5				
I7		r3	r3	r3			
I8		r6	r6				
I9		r5	r5	r5			
I10		r6	r6	r6			

[0.5 Marks]

Since the LR(1) table **contains multiple S-R conflicts**, hence it is **not a LR(1) grammar**.

Part (b):

[2 Marks]



[2 Marks]

	–	+	e	$	S	B	E
I0	s3				1	2	
I1				Accept			
I2		r7	s96, r7	r2			4
I3		r7	s96, r7	r7			5
I4		s7	s108				
I5		r4	s108, r4	r4			
I7		r3	r3	r3			
I96		r5	r5	r5			
I108		r6	r6	r6			

[0.5 Marks]

Since the LALR(1) table **contains multiple S-R conflicts**, hence it is **not a LALR(1) grammar**.

<u>Question-3 (Intermediate Code Generation):</u> **[20 Points]**

Assume that your programming language supports control flow instructions for the “For Loop” using the following grammar

foreach-statement → foreach(ID in [START..END]) S ;

Semantics are as follows:

1. ID gets initialised to constant values START and executes S iteratively as long as the ID<=END.
2. At the end of each iteration ID gets incremented by 1.

Part-(a): Write down the semantic rules to generate the three address codes as intermediate representation.

Hints: (1) You can use the backpatch function discussed in the class. (2) You can introduce the markers (M, N, etc) into the above grammar with suitable epsilon transitions to pick up index of next quadruple

<u>Part-(a): 17 Marks</u>

```
S→  foreach(ID in [M1 START..M2 END] ) N1 S N2 ;
     {
                   backpatch(M2.truelist, N1.quad);
                   backpatch(N2.nextlist, M2.quad);
                   S.nextlist=M2.falselist;
      } // 5 Marks

M1→ ∊  {emit(“ID=START”); }  //1 Mark

M2→ ∊  {M2.quad = nextquad;
          M2.truelist = makelist(nextquad);
          M2.falselist = makelist(nextquad+1);
          emit(“if(ID<=END) goto __”);
          emit(“goto ______ “);
          }    // 5 Marks
N1→ ∊  {N.quad = nextquad }  //1 Mark

N2→ ∊  {emit(“id=id+1”);
         N2.nextlist = makelist(nextquad);
          emit(“goto ___”);
         }   // 5 Marks
```

Part-(b): 3 Marks

Generate the three address codes for the following statement with the above semantics.

```
foreach(i in(1..20))
        j=j*b+d;
```

```
100: i=1   //0.5 Marks
101: if i<20 goto 103  //0.5 Marks
102: goto 108  //0.5 Marks
103: t1=j*b    //Step 103 to 105 carries 0.5 Marks
104: t2=t1+d
105: j=t2
106: i=i+1 //0.5 Marks
107: goto 101  //0.5 Marks
108:
```

Question-4 (Code Generation): [30 Points]

Assume that an architecture supports the ternary operations ⊕, ⊙, ⊖. Each of which accepts 3 operators.

Example usage of the operators is: OP(a,b,c), where OP can be ⊕, ⊙, or ⊖. These operators perform some computation based on the values provided by a, b, and c.

Assume that we have the following target machine model.

Instruction	**Semantics**
Store m, r	Stores the contents of register r to the memory location m
Load r, m	Loads the contents of the memory location m to the register r
OP r2, r2, m, r1	Performs the ternary operation of register r2, memory operand m and register operand r1. The result of OP r2, m, r1 is stored in r2. Here OP can be ⊕, ⊙, or ⊖
OP r3, r3, r2, r1	Performs the ternary operation of register r3, register r2, and register r1. The result of OP r3, r2, r1 is stored in r3. Here OP can be ⊕, ⊙, or ⊖

(a) Modify the labelling algorithm that we discussed in the class to calculate the minimum number of registers required to compute an expression tree supporting the above target machine model.

(b) Modify the Sethi-Ullman Algorithm to generate the code. Discuss various cases of your algorithm for code generation. You can assume that your target has a sufficient number of registers, so you don't need to consider the case of moving to temporary locations.

(c) Use the modified labelled algorithm and the code generation algorithm to generate the code for the following expression

$$\odot(\oplus(a,b,c),d,\oplus(\odot(e,f,g),\ominus(h,i,j),k))$$

You must draw the expression tree and list the label for each node in the tree, and the final sequence of assembly instructions that are generated for the expression.

Part-(a): **5 Marks**

Since the first and third operands of the ternary operation must be registers, we require 1 register for each corresponding leaf node to load the contents of the memory location to the register. Hence the modified algorithm is as follows.

1. Label left most leaf and right most leaf node by 1. **0.5 Marks**
2. Label the middle leaf node as 0. **0.5 Marks**
3. If the labels of the children of a node n are l_1 and l_2 and l_3, in the same order from left to right

label(l) = $\max(l_1, l_2, l_3)$ if $l_1 \neq l_2 \neq l_3$ **1 Mark**

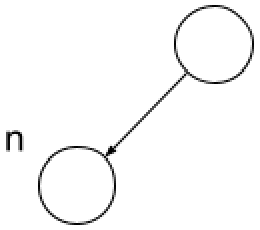
= l_1+2 if $l_1=l_2=l_3$ **1 Mark**

= $\max(l_i+1, l_k)$ if $l_i=l_j$ where $i\neq j$ where $i\neq j$, $l_i+1\neq l_k$ **1 Mark**

= l_1+2 if $l_i=l_j$ where $i\neq j$ where $i\neq j$, $l_i+1= l_k$ **1 Mark**

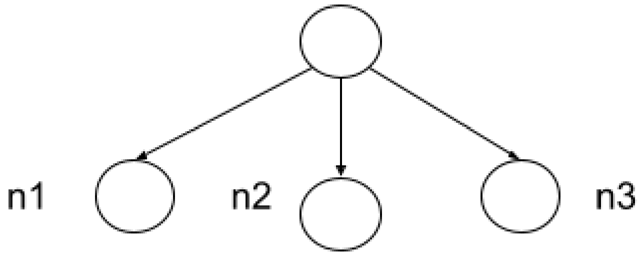
Part-(b): **14 Marks (each case 2 Marks)**

Case-1: n is a left most leaf:



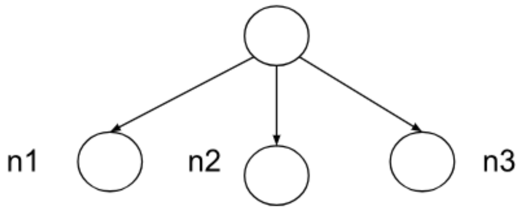
gen(top(rstack) ← n.name)

Case-2: n is a right most leaf (in the below n3 is leaf)



```
gencode(n1)
R=pop(rstack)
gen( top(rstack ) ← n3.name)
R  ← OP(R, n2.name, top(rstack)
Push(rstack, R)
```

Case-3: left most child is heavier than or as heavy as other nodes (i.e, n1 >= n2, n3)



```
gencode(n1)
R1=pop(rstack)
gencode(n2)
R2=pop(rstack)
gencode(n3)
R1  ← OP(R1, R2,  top(rstack))
push(rstack, R2)
push(rstack, R1)
```

Case-4: n2 >= n3 > n1 (where i and j are different)

```
R1=pop(rstack)
gencode(n2)
R2=pop(rstack)
gencode(n3)
R3=pop(rstack)
push(rstack, R1)
gencode(n1)
top(rstack)  ← OP(top(rstack), R2, R3)
```

```
R1=pop(rstack)
push(rstack, R3)
push(rstack, R2)
push(rstack, R1)
```

Case-5: n3 > n2 > n1 (where i and j are different)

```
R1=pop(rstack)
gencode(n3)
R3=pop(rstack)
gencode(n2)
R2=pop(rstack)
push(rstack, R1)
gencode(n1)
top(rstack)  ← OP(top(rstack), R2, R3)
R1=pop(rstack)
push(rstack, R3)
push(rstack, R2)
push(rstack, R1)
```

Case-6: n2 > n1 > n3

```
R1=pop(rstack)
gencode(n2)
R2=pop(rstack)
push(R1)
gencode(n1)
R1=pop(rstack)
gencode(n3)
R1  ← OP(R1, R2, top(rstack))
R1=pop(rstack)
push(rstack, R2)
push(rstack, R1)
```

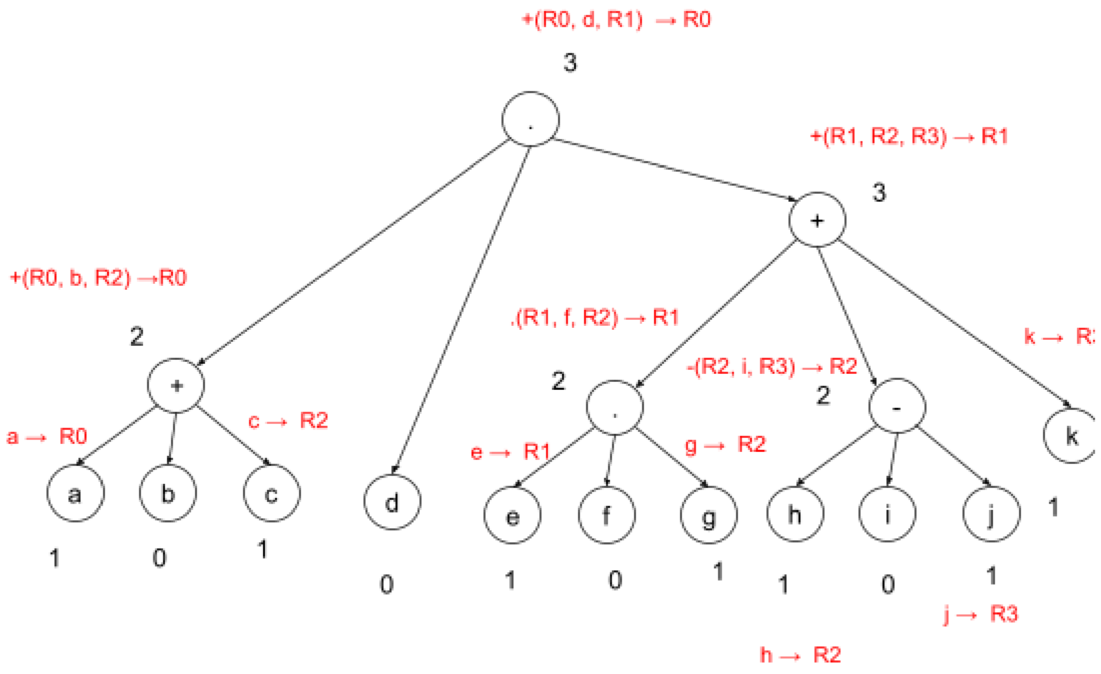
Case-7: n3 > n1 > n2

```
R1=pop(rstack)
gencode(n3)
R3=pop(rstack)
push(R1)
gencode(n1)
R1=pop(rstack)
gencode(n2)
```

R1 ← OP(R1, top(rstack),R3)
R1=pop(rstack)
push(rstack, R3)
push(rstack, R1)

Part-(c): 11 Marks

Marks distribution is given below.
6 Marks for codes at the nodes (0.5 Mark for each correct code at the node)
Labelling
1 Mark for labelling the leaf nodes
0.5 Mark for labelling the intermediate nodes
0.5 Mark for the root node labelling



Sequence of code: //3 Marks
e → R1
g → R2
.(R1, f, R2) → R1
h → R2
j → R3
-(R2, i, R3) → R2
k → R3
+(R1, R2, R3) → R1
a → R0
c → R2
+(R0, b, R2) → R1
.(R0, d, R1) → R0

Question-5 (Machine Independent Optimizations): [15 Points]

Consider the following snippet of the code.

```
a=1;
b= a+b;
e= c+d;
if((a<b) or (c>d) )
{
        j=4;
        while ((e<f ) and (a!= (e+d))) {
                e=e+f ;
                a=c+d ;
        }
}
else
{
        b=c+d;
}
```

(a) Generate the three-address code representation for the above program. Assume that the implementation uses short circuit evaluation for Boolean operators. You can assume the precedence of the associativity follows that of C language.

(b) Apply the loop invariant code motion and global common sub expression elimination to the above three address codes.

(c) Apply the copy-propagation for the code generated for the code generated in Part(b) and show the optimized code.

Solution: Part (a) Three -address code representation [8 marks]

```
101: a = 1
102: t1 = a + b
103: b = t1
104: t2 = c + d
105: e = t2
106: if a < b goto 110
107: goto 108
108: if c > d goto 110
109: goto 122
110: j = 4
        // loop test start
111: if e < f goto 113
```

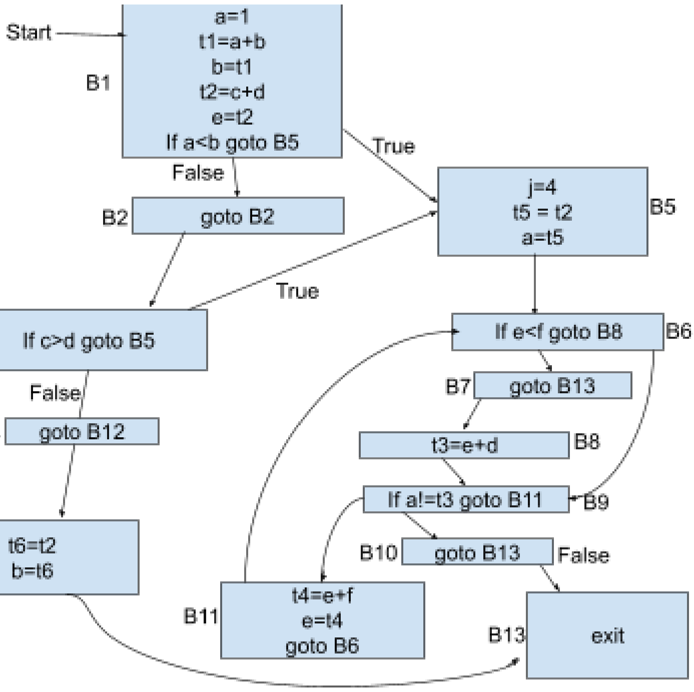
```
112: goto 123                  // exit loop
113: t3 = e + d
114: if a != t3 goto 116
115: goto 123                 // exit loop
116: t4 = e + f
117: e = t4
118: t5 = c + d
119: a = t5
120: goto 111                 // repeat loop
121: t6 = c + d
122: b = t6
123:
```

Part (b)

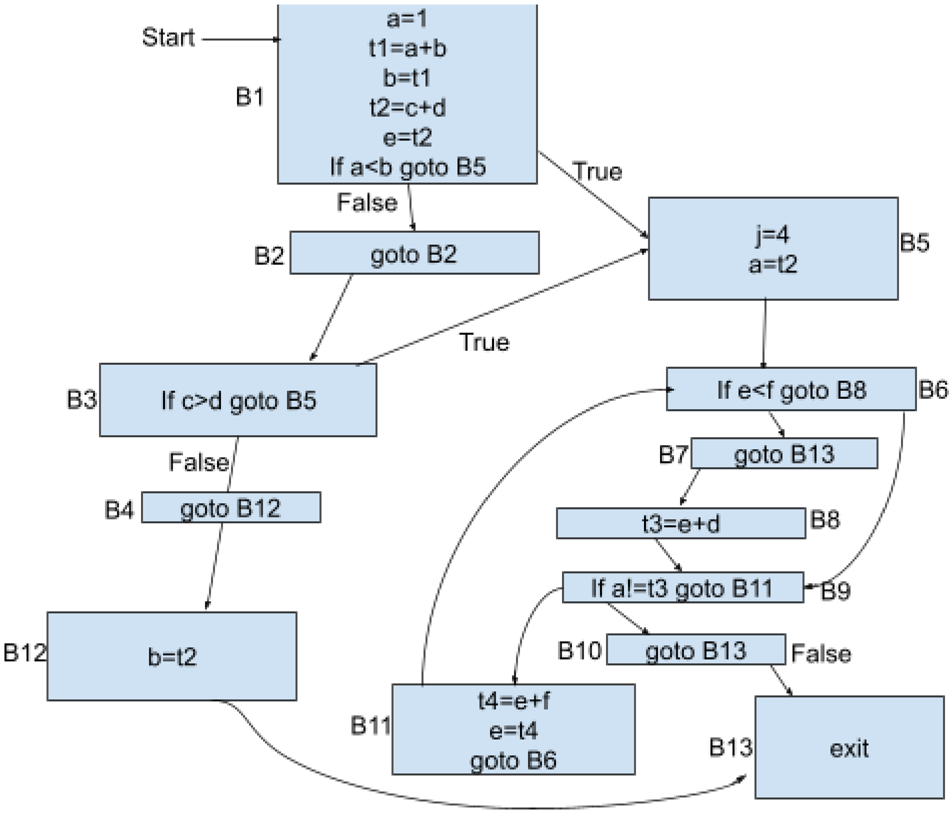
Loop invariant code [1 mark]

```
101: a = 1
102: t1 = a + b
103: b = t1
104: t2 = c + d
105: e = t2
106: if a < b goto 110
107: goto 108
108: if c > d goto 110
109: goto 122
110: j = 4
111: t5 = c + d
112: a = t5
         // loop test start
113: if e < f goto 116
114: goto 123                  // exit loop
115: t3 = e + d
116: if a != t3 goto 118
117: goto 123                 // exit loop
118: t4 = e + f
119: e = t4
120: goto 113                 // repeat loop
121: t6 = c + d
122: b = t6
123:
```

Global common sub-expression [3 marks]



Part (C) Copy propagation [3 marks]


Start
B1
a=1
t1=a+b
b=t1
t2=c+d
e=t2
If a<b goto B5
True
False
B2
goto B2
B5
j=4
a=t2
True
B3
If c>d goto B5
B6
If e<f goto B8
False
B7
goto B13
B4
goto B12
B8
t3=e+d
If a!=t3 goto B11
B9
B12
b=t2
B10
goto B13
False
B11
t4=e+f
e=t4
goto B6
B13
exit