Question 1 (4 + 4 + 2 = 10 points)

Consider the following segment of three-address code.

```
1:  x = z - 2
2:  y = z * 2
3:  t1 = x + 60
4:  if y < t1 goto 9
5:  x = x - 1
6:  y = y - 1
7:  goto 11
8:  z = z * 2
9:  x = x +1
10: y = y + 1
11: z = x + y
12: if z < 30 goto 2
13: y = x
```

1. Indicate the basic blocks of the above code segment.

   B1: $\langle 1 \rangle$

   B2: $\langle 2, 3, 4 \rangle$

   B3: $\langle 5, 6, 7 \rangle$

   B4: $\langle 8 \rangle$

   B5: $\langle 9, 10 \rangle$

   B6: $\langle 11, 12 \rangle$

   B7: $\langle 13 \rangle$

2. Draw the flow graph for the above segment.

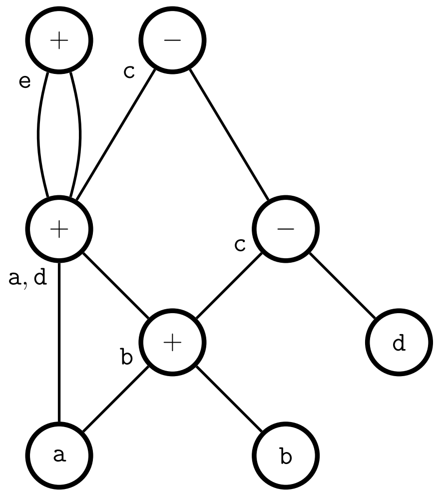
   Entry $\longrightarrow$ **B1** $\longrightarrow$ **B2** $\longrightarrow$ **B3** $\longrightarrow$ **B6** $\longrightarrow$ **B7** $\longrightarrow$ Exit
   B2 $\longrightarrow$ **B5** $\longrightarrow$ **B6**
   B4 $\longrightarrow$ **B5**

3. Just by inspecting the graph, suggest at least one possible modification to the code which will optimize performance.

   Remove instruction 8.

Question 2 (3 + 3 + 3 = 9 points)

Consider the following DAG representation of a basic block of three-address code.



1. Write a segment of three-address code for which the above DAG is a representation.

   Answer.

   ```
   1. b = a + b
   2. c = b - d
   3. d = a + b
   4. a = a + b
   5. c = a - c
   6. e = a + a
   ```

2. Suppose that the cost of a three-address instruction is 1 + the number of memory references needed to execute the instruction, and that each constant or variable operand requires a separate fetch operation from memory, unless stored in a register. Compute the cost of the code segment given as answer to part (1), showing the cost of each instruction. Assume that variables `a`, `c`, and `e` are stored in registers.

   $Cost(1) = 1 + 1 + 1 + 1 + 1 = 5$
   $Cost(2) = 1 + 1 + 1 + 1 + 1 = 5$
   $Cost(3) = 1 + 1 + 1 + 1 + 1 = 5$
   $Cost(4) = 1 + 1 + 1 = 3$
   $Cost(5) = 1$
   $Cost(6) = 1$
   Total cost is 20.

3. Optimize the three-address code segment as much as possible, if you know that only variable `e` is live on exit from the block. What is the cost of the optimized code?

   The following code is equivalent to the original one with a cost of 5.

```
a = a + a
a = a + b
e = a + a
```

Question 3 (16 points)

Consider the following SDD for simple numerical types; T, B, and C are non-terminals.

$$
\begin{array}{lcll}
T & \longrightarrow & BC & C.t = B.type; C.w = B.width \\
 & & & T.type = C.type; T.width = C.width \\
B & \longrightarrow & \textbf{int} & B.type = integer; B.width = 4 \\
B & \longrightarrow & \textbf{float} & B.type = float; B.width = 8 \\
C & \longrightarrow & [\textbf{num}]C_1 & C_1.t = C.t; C_1.w = C.w \\
 & & & C.type = array(\textbf{num}.value, C_1.type) \\
 & & & C.width = \textbf{num}.value \times C_1.width \\
C & \longrightarrow & \varepsilon & C.type = C.t; C.width = C.w
\end{array}
$$

Draw the *annotated parse tree* and associated *dependency graph* (indicating the values of all attributes of non-terminal occurrences) for the input string `int`[3][3][2].

Check the example on Slide 14 of Lecture 9.

Question 4 (20 points)

Complete the following SDD which generates code for short-circuit evaluation of Boolean expressions. B, D, and C are non-terminals.

$B \longrightarrow B_1$ `||` D $\qquad B_1.true = B.true; B_1.false = newlabel()$

$D.true = B.true; D.false = B.false$

$B.code = B_1.code \circ label(B_1.false) \circ D.code$

$B \longrightarrow D$

$D \longrightarrow D_1$ `&&` C

$D \longrightarrow C$

$C \longrightarrow$ `(`B`)`

$C \longrightarrow$ `!`B

$C \longrightarrow$ **true**

$C \longrightarrow$ **false**

Check Slides 27 through 33 of Lecture 10.

Question 5 (12 points)

In Common Lisp, a non-empty list of integers is internally represented by a dotted pair $(f \cdot r)$, where f is an integer and r is a list of integers; empty lists are internally represented by the constructor `nil`. Write an SDT (*not an SDD*) to translate from the internal Lisp representation of integer lists to the more readable representation where a list of integers is denoted by a (possibly empty) comma-separated sequence of integers between parentheses. The output need only be printed (and not held as the value of an attribute); you may assume a function $print(s)$ that prints a string s. Your SDT should account for the following sample input-output pairs.

Input	**Output**
`(1 . (2 . nil))`	`(1, 2)`
`nil`	`()`
`(1 . nil)`	`(1)`

You may assume that a lexical analyzer scans the input so that your SDT receives a stream of symbols in which every occurrence of an integer has been replaced by the token **int**. Values of different **int** tokens may be retrieved via the call **int**.$lexval()$.

$$
\begin{array}{lcl}
L & \longrightarrow & \{\texttt{print}("(")\}\ (F.R)\ \{\texttt{print}(")")\} \\
L & \longrightarrow & \texttt{nil}\ \{\texttt{print}("()")\} \\
R & \longrightarrow & \{\texttt{print}(",")\}\ (F.R) \\
R & \longrightarrow & \texttt{nil} \\
F & \longrightarrow & \textbf{int}\ \{\texttt{print}(\textbf{int}.lexval())\}
\end{array}
$$

Question 6 (12 + 11 + 6 = 29 points)

Consider the following grammar G for simple integer variable declarations, where C and D are non-terminals.

$$\begin{array}{lll} 1. & D & \longrightarrow & \textbf{int}\ C\ \textbf{id};\ D \\ 2. & D & \longrightarrow & \varepsilon \\ 3. & C & \longrightarrow & [\ \textbf{num}\ ]\ C \\ 4. & C & \longrightarrow & \varepsilon \end{array}$$

1. Draw the state diagram of the LR(1) DFA for G.

2. Construct the canonical LR(1) parsing table for G.

3. Trace the operation of the LR parsing algorithm on G and input

int [**num**] **id** ;

Make sure you show the successive contents of the stack and the corresponding part of the input string which is yet to be read.

Question 7 ($3 + 3 = 6$ points)

1. Show that the grammar G of Question 6 is an LALR grammar. *You do not need to construct the LALR DFA.*

2. Show that G is an SLR(1) grammar. *You do not need to construct the LR(0) DFA.*