1. Consider the flow graph in Dragon book, Figure 9.10 (also given as follows).

(a) Compute Gen and Kill sets for each block in the flow graph.

- **Gen** and **Kill** sets for each block:
  - Gen\[B1\] = \{(1),(2)\}
  - Kill\[B1\] = \{(8),(10),(11)\}
  - Gen\[B2\] = \{(3),(4)\}
  - Kill\[B2\] = \{(5),(6)\}
  - Gen\[B3\] = \{(5)\}
  - Kill\[B3\] = \{(4),(6)\}
  - Gen\[B4\] = \{(6),(7)\}
  - Kill\[B4\] = \{(4),(5),(9)\}
  - Gen\[B5\] = \{(8),(9)\}
  - Kill\[B5\] = \{(2),(7),(11)\}
  - Gen\[B6\] = \{(10),(11)\}
  - Kill\[B6\] = \{(1),(2),(8)\}

(b) Compute the In and Out sets for each block in the flow graph.

**Round 0. Initialization:**

<table>
<thead>
<tr>
<th>In[B1]</th>
<th>Out[B1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>{1,2}</td>
</tr>
<tr>
<td>∅</td>
<td>{3,4}</td>
</tr>
<tr>
<td>In[B3]</td>
<td>Out[B3]</td>
</tr>
<tr>
<td>∅</td>
<td>{5}</td>
</tr>
<tr>
<td>∅</td>
<td>{6,7}</td>
</tr>
<tr>
<td>In[B5]</td>
<td>Out[B5]</td>
</tr>
<tr>
<td>∅</td>
<td>{8,9}</td>
</tr>
<tr>
<td>∅</td>
<td>{10,11}</td>
</tr>
</tbody>
</table>

**Round 1:**

| In[B1] = ∅ | Out[B1] = \{1,2\} |
| In[B3] = Out[B2] ∪ Out[B4] = \{1,2,3,4,6,7,8,9\} | Out[B3] = \{1,2,3,5,7,8,9\} |
| In[B4] = Out[B3] = \{1,2,3,5,7,8,9\} | Out[B4] = \{1,2,3,6,7,8\} |
| In[B5] = Out[B3] = \{1,2,3,5,7,8,9\} | Out[B5] = \{1,3,5,8,9\} |

**Round 2:**

| In[B1] = ∅ | Out[B1] = \{1,2\} |
| In[B2] = \{1,2,3,5,8,9\} | Out[B2] = \{1,2,3,4,8,9\} |
| In[B3] = \{1,2,3,4,6,7,8,9\} | Out[B3] = \{1,2,3,5,7,8,9\} |
| In[B4] = \{1,2,3,5,7,8,9\} | Out[B4] = \{1,2,3,6,7,8\} |
| In[B5] = \{1,2,3,5,7,8,9\} | Out[B5] = \{1,3,5,8,9\} |
| In[B6] = \{1,3,5,8,9\} | Out[B6] = \{3,5,9,10,11\} |
(c) Perform common subexpression elimination.

<table>
<thead>
<tr>
<th>In[B1]</th>
<th>Out[B1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>{c = a+b, d = c-a}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In[B3]</th>
<th>Out[B3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{c = a+b, d = c-a}</td>
<td>{c = a+b, d = c-a, d = b+d}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{c = a+b, φ = c-a, d = b+d}</td>
<td>{c, d = a+b, φ = c-a, e = c+1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In[B5]</th>
<th>Out[B5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{b = a+b, φ = c-a}</td>
<td>{b = a+b, e, φ = c-a}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{b = a+b, φ = c-a}</td>
<td>{a = b+d, b = a-d}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Round 2:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>In[B1]</th>
<th>Out[B1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>{c = a+b, d = c-a}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In[B3]</th>
<th>Out[B3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[B2] ∩ Out[B4] = {c = a+b, d = c-a}</td>
<td>{c = a+b, d = c-a, d = b+d}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{c = a+b, φ = c-a, d = b+d}</td>
<td>{c, d = a+b, φ = c-a, e = c+1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In[B5]</th>
<th>Out[B5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{b = a+b, e, φ = c-a}</td>
<td>{b = a+b, φ = c-a}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{b = a+b, φ = c-a}</td>
<td>{a = b*d, b = a-d}</td>
</tr>
</tbody>
</table>

**Statements change for common sub expression elimination:**

**B2**  
(3) \( c = a + b \Rightarrow t_1 = a + b; \ c = t_1 \)  
(4) \( d = c - a \Rightarrow t_2 = c - a; \ d = t_2; \)

**B4**  
(6) \( d = a + b \Rightarrow d = t_1 \)

**B5**  
(8) \( b = a + b \Rightarrow b = t_1 \)  
(9) \( e = c - a \Rightarrow e = t_2 \)
2. Consider the following three address code in a basic block.

(a) Perform copy propagation.

\[
\begin{align*}
(1) & \quad t1 = j - 1 \\
(2) & \quad t2 = 4 \ast t1 \\
(3) & \quad temp = A[t2] \\
(4) & \quad t3 = j \\
(5) & \quad t4 = t3 + 1 \\
(6) & \quad t5 = 4 \ast t3 \\
(7) & \quad t6 = A[t5] \\
(8) & \quad t7 = j - 1 \\
(9) & \quad t8 = 4 \ast t7 \\
(10) & \quad A[t8] = A[t5] \\
(11) & \quad t9 = j \\
(12) & \quad t10 = t9 + 1 \\
(13) & \quad t11 = 4 \ast t9 \\
(14) & \quad A[t11] = temp
\end{align*}
\]

(b) Perform liveliness analysis and dead code elimination based on the results from (a). Assume that after the basic block, array \(A[i]\), for all \(i\), are alive and no other variables are alive.

<table>
<thead>
<tr>
<th>(Round 1) In[1]=Out[1]= \emptyset</th>
</tr>
</thead>
<tbody>
<tr>
<td>In[8]= {3, 4, 7}, Out[8]= {3, 4, 7}</td>
</tr>
<tr>
<td>In[10]= {3, 4, 7}, Out[10]= {3, 4, 7, 10}</td>
</tr>
<tr>
<td>In[12]= {4, 10, 11}, Out[12]= {4, 7, 10, 11}</td>
</tr>
<tr>
<td>In[14]= {4, 10, 11}, Out[14]= {4, 7, 10, 11, 14}</td>
</tr>
</tbody>
</table>

(4) (5) (7) (11) (12) are dead code, because the live sets “before” (4) (5) (7) (11) (12) does not contain the variables \(t3, t4, t6, t9, t10\) being defined in (4) (5) (7) (11) (12), respectively.

Note: Array \(A\) is a special case, since it is an array, we do not know the exact address being defined or used. So, In (a), A def kills A def; In (b), A use is not killed by A def, but, for example, \(A[5]\) kills \(A[5]\).
3. Consider the following flow graph. Perform constant propagation and folding using MFP.

Round 1.
In[B1] = ∅
Out[B1] = { b=3, c=5, d=10, g=6 }
Out[B2] = { a=9, b=3, c=5, d=10, g=6 }
In[B3] = { a=9, b=3, c=5, d=10, g=6 }
Out[B3] = { a=9, b=3, c=6, d=9, g=30 }
In[B4] = { a=9, b=3, c=5, d=10, g=6 }
Out[B4] = { a=9, b=3, c=7, d=8, g=30 }
In[B5] = Out[B3] \∩ Out[B4] = { a=9, b=3, g=30 }
Out[B5] = { a=12, b=3, g=30, h=42 }
In/Out[B6] = Out[B5]

Round 2.
In[B1] = ∅
Out[B1] = { b=3, c=5, d=10, g=6 }
Out[B2] = { a=9, b=3 }
In[B3] = { a=9, b=3 }
Out[B3] = { a=9, b=3 }
In[B4] = { a=9, b=3 }
Out[B4] = { a=9, b=3 }
In[B5] = Out[B3] \∩ Out[B4] = { a=9, b=3 }
Out[B5] = { a=12, b=3 }
In/Out[B6] = Out[B5]
4(a) Perform liveliness analysis. Show the Live sets at each point of the CFG. Identify the dead code.

Dead code: read (a) in B1 and a := c + d in B4.

It is ok not to consider read(a) as dead code because it may have some desired side effect.

4(b) Assume that the system has five registers. Construct the interference graph. Perform color assignment. Determine the register for each variable.

5-color: All nodes have < 5 edges. It is 5-colorable.

The registers for the variables are listed as follows:

a: R4  b: R4  c: R3  
d: R2  f: R1  v: R0
4(c) Assume that the system has only three registers. Perform color assignment and determine the register for each variable. In case it is necessary to spill, spill and revise the code and then do color assignment.

3-color: Remove “a” first ⇒ Cannot find a node with 2 edges ⇒ Need to spill ⇒ Choose to spill v

The interference graph after removing a and spilling v is given below (below left).

Still cannot find a node with 2 edges ⇒ Need to spill again ⇒ Choose to spill c.

Now the graph is 3-colorable.

Next we need to check whether the spilled nodes can have their registers when loaded back.

Change the code, perform liveness analysis again and construct the new interference graph (above right). The new interference graph is three colorable.

So the registers assigned for the variables are listed below:

b: R0  f: R1  d: R2  a: R2  c,v: spilled

t1: R0  t2: R1  t3: R2  t4: R1
5. Consider the following attribute grammar.

\[
\begin{align*}
S \rightarrow & \text{id} \,:= \, E \quad \{ \, p := \text{lookup (id.name)}; \, \text{emit (p.place} \, \text{:=} \, E\text{.place); } \}
E \rightarrow & \, E_1 \, + \, T \quad \{ \, E\text{.place} := \text{newtemp}; \, \text{emit (E.place} \, \text{:=} \, E_1\text{.place} \, \text{+} \, T\text{.place); } \}
E \rightarrow & \, T \quad \{ \, E\text{.place} := T\text{.place; } \}
T \rightarrow & \, T_1 \, * \, F \quad \{ \, T\text{.place} := \text{newtemp}; \, \text{emit (T.place} \, \text{:=} \, T_1\text{.place} \, * \, F\text{.place); } \}
T \rightarrow & \, F \quad \{ \, T\text{.place} := F\text{.place; } \}
F \rightarrow & \, (E) \quad \{ \, F\text{.place} := E\text{.place; } \}
F \rightarrow & \, \text{id} \quad \{ \, p := \text{lookup (id.name)}; \, \text{emit (F.place} \, \text{:=} \, p\text{.place); } \}
F \rightarrow & \, \text{num} \quad \{ \, \text{emit (F.place} \, \text{:=} \, \text{num.value); } \}
\end{align*}
\]

Given input: \( x := a + b + c + ( (d + e) \, * \, (f + g) + (h + i) \, * \, (j + k) ) \, * \, (l + m) \, * \, (n + o) + p \).

Generate the three address code using the attribute grammar for the input. Construct the computation tree from the three address code and allocate registers for the computation. Change the tree-based register allocation algorithm to only allocate registers to the left leave node, not the right leave node. You can perform these steps in your mind.

\[
\begin{align*}
\text{(a) Determine the number of registers required in the machine code generated for the input statement.}
\text{(b) Determine the register allocation for the computation.}
\text{(c) Generate the machine code using four instructions:}
\end{align*}
\]

\[
\begin{align*}
\text{load register id/num:} & \quad \text{register} = \text{id/num} \\
\text{add register1 register2 register3/id/num:} & \quad \text{register1 = register2 + register3/id/num} \\
\text{mul register1 register2 register3/id/num:} & \quad \text{register1 = register2 * register3/id/num} \\
\text{store register id:} & \quad \text{id = register} \\
\end{align*}
\]

\[
\begin{align*}
\text{load r0 d} & \quad \text{load r1 l} & \quad \text{load r0 a} \\
\text{add r0 r0 e} & \quad \text{add r1 r1 m} & \quad \text{add r0 r0 b} \\
\text{load r1 f} & \quad \text{add r2 n} & \quad \text{add r0 r0 c} \\
\text{add r1 r1 g} & \quad \text{add r2 r2 o} & \quad \text{add r0 r0 d} \\
\text{mul r0 r0 r1} & \quad \text{mul r1 r1 r2} & \quad \text{add r0 r0 e} \\
\text{load r1 h} & \quad \text{add r2 r1 p} & \quad \text{add r0 r0 f} \\
\text{add r1 r1 i} & \quad \text{add r2 r2 k} & \quad \text{add r0 r0 g} \\
\text{load r2 j} & \quad \text{mul r2 r1 r2} & \quad \text{add r0 r0 h} \\
\text{add r2 r2 k} & \quad \text{add r0 r0 i} & \quad \text{store r0 x} \\
\text{mul r0 r0 r2} & \quad \text{add r0 r0 j} & \quad \text{add r0 r0 k} \\
\text{add r0 r0 r2} & \quad \text{add r0 r0 l} & \quad \text{add r0 r0 m} \\
\end{align*}
\]
6. Consider the following three address code.

```plaintext
t1 := j
 t1 := t1 * dim2
 t1 := t1 + k
 t1 := t1 * w
 t2 := j * k
 t3 := t3 * dim2
 t3 := t3 + k
 t3 := t3 * w
 t4 := B [ t3 ]
 t5 := t2 + t4
 A [ t1 ] := t5
```

(a) Construct a dag from the three address code such that subexpressions and redundant variables are eliminated.

(b) Perform dag based register allocation and scheduling. Generate code based on the register allocation and instruction schedule you derived. The 6 instructions are given below. Your code should make sure that the content of array A is permanently modified.

- load register id/num : register = id/num
- store register id : id = register
- loadarray register1 id register2 : register1 = id[register2]
- storearray register1 id register2 : id[register2] = register1
- add register1 register2 register3/num : register1 = register2 + register3/num
- mul register1 register2 register3/num : register1 = register2 * register3/num
Stack
3, 4
4, 5
6, 7
7, 8, 9
9, 10, 11
empty

Paths
P1: red (1, 5, 6, 8, 11)
P2: blue ((8), 10)
P3: orange (1, 2, 4)
P4: violet ((4), 9)
P5: green ((6))
P6: brown ((5), 7)
P7: turquoise ((2), 3)

Re-number the operations
red: r0; blue: r2; orange: r1
violet: r3; green: r3
brown: r2; turquoise: r2

Some orderings due to fusion is already satisfied

Fusion
P2→P4: P4 flow into P2, no
P4→P2: then 6→8, no
P4, P5 should fuse
P6, P7 can fuse with P2 or P4

Scheduling constraint
Fusion constraint
7. Consider the following instruction set (only those for basic blocks). Some costs for the instructions is unreasonable, just for the purpose of practicing the instruction selection algorithms.

(a) Draw the tiles for operators aload, astore, addx, mulx. (The tiles for the remaining operators can be found in the notes and in the book.)

(b) Draw the instruction tree for the basic block given above.

(c) Tile the tree using the maximal munching algorithm. Assign registers using the tree based algorithm. Generate machine code.
load R2 j
addc R2 R2 1
mulx R2 R2 d2
adds R2 R2 k
mulx R2 R2 w
load R1 &B
aload R1 R1 R2
load R2 &A
load R3 j
mulx R3 k
mulx R3 w
aload R3 R2 R3
add R2 R1 R3
load R2 j
addc R2 R2 1
mulx R2 R2 d2
adds R2 R2 k
mulx R2 R2 w
load R1 &A
astore R3 R1 R2

(d) Tile the tree using the dynamic programming algorithm.
8. Consider the following program.

```
for (i=2; i<=n; i++)
    a[i] = TRUE;

count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) {
        count++;
        for (j=2*i; j<=n; j = j+1)
            a[j] = FALSE;
    }
```

(a) Translate the program into three address code as defined in Section 6.2, dragon book.
(b) Identify all basic blocks in your three address code.
(c) Build the flow graph for the three address code.

(a) three address code

```
(1) i := 2
(2) if i > n goto (7)
(3) a[i] := TRUE
(4) t2 := i+1
(5) i := t2
(6) goto (2)
(7) count := 0
(8) s := sqrt(n)
(9) i := 2
(10) if i > s goto (23)
(11) if a[i] != TRUE goto (20)
(12) t4 := count +1
(13) count := t4
(14) j := 2*i
(15) if j > n goto (20)
(16) a[j] := FALSE
(17) t6 := j+1
(18) j := t6
(19) goto (15)
(20) t7 := i+1
(21) i := t7
(22) goto (10)
(23) exit
```

(b) basic blocks

- B1: 1
- B2: 2
- B3: 3-6
- B4: 7-9
- B5: 10
- B6: 11
- B7: 12-14
- B8: 15
- B9: 16-19
- B10: 20-22
- B11: 23
(d) Build the dominator tree and identify the back edges in your flow graph in (c).

Back edges
(10,5)
(9,8)
(3,2)

(e) Find the entry node and the set of nodes in the natural loop associated with each back edge identified in (d).

back edge (10,5):
  entry node: 5
  loop nodes: {5,6,7,8,9,10}
back edge (9,8):
  entry node: 8
  loop nodes: {8,9}
back edge (3,2):
  entry node: 2
  loop nodes: {2,3}
9. Consider the following flow graph.
(a) Identify and mark define-use links within the loop based on data flow analysis results.

(b) Identify all the loop invariants. Assume that all the loop invariants can be moved out of the loop. So you need to find loop invariants repetitively till a fixed point is reached. In each round, you need to pretend to move out those loop invariants you found in the previous rounds.

**Round 1:**
- **B2:** Statement “N = 100” is a loop invariant, pretend to move it out of the loop
- **B3:** Statement “M = 1000” is a loop invariant, pretend to move it out of the loop
- **B3:** Statement “s1 = 2” is a loop invariant, pretend to move it out of the loop

**Round 2:**
- **B4:** Statement “s2 = N + 2” now is a loop invariant, pretend to move it out of the loop
- **B5:** Statement “t3 = N” now is a loop invariant, pretend to move it out of the loop
(c) For each loop invariant, determine whether it can actually be moved out of the loop. If so, move it out of the loop. If not, state the reason why it cannot be moved out. Generate the new code after code motion.

Round 1:
B2: “N = 100”: Satisfies all 3 criteria, move it out of the loop
    “M = 1000”: There are multiple definitions of M in the loop, cannot be moved out
    “s1 = 2”: There are multiple definitions of s1 in the loop, cannot be moved out
B3: B3 does not dominate the exit, no statement can be moved out

Round2:
B4: “s2 = N + 2”: Is still a loop invariant, and satisfies all 3 criteria, move it out of the loop
B5: B5 does not dominate the exit, no statement can be moved out
10. Consider the following flow graph. Perform strength reduction for the statements in the loop.

```
| t1 = 0   | B1 |
| read (x, y) |

| a := x * t1 | B2 |
| t2 := t1 * 3 + 5 |
| b := y * t2 |
| c := t2 + 5 |
| d := c * b |
| t1 := t1 + 2 |
```

“t1 = t1 + 2” in B2 is a basic IV
“t2 := t1*3+5” B2 is in the family of t1.

We can perform strength reduction for t2 and the revised code is given below:

```
| t1 = 0   | B1 |
| read (x, y) |

| st2 = t1*3+5 |

| a := x * t1 | B2 |
| t2 := st2 |
| b := y * t2 |
| c := t2 + 5 |
| d := c * b |
| t1 := t1 + 2 |
| st2 := st2 + 6 |
```

11. Consider the following function with tail recursion. Eliminate the tail recursion. Note: You should follow the tail recursion elimination rule, not to change the code because you know what it is doing.

```java
Function fib (a, b, n)
{ if (n == 0) return b;
  a := a + b;
  return (fib (b, a, n–1));
}
```

```java
Function fib (a, b, n)
{ start: if (n == 0) return b;
  a := a + b;
  t1 = b;  t2 := a;  t3 := n – 1;  a := t1;  b := t2;  n := t3;  go to start;
}
```
12. Consider the following CFG. In the CFG, p, q, r, s are pointers.
(a) Perform alias analysis for the CFG. Give the graphical representation for the Out sets.
(b) Perform constant propagation/folding in your mind and change the code after the analysis.

Out[B1] = {(p,a), (q,b)}
In[B2] = {(p,a), (q,b)}
Out[B2] = {(p,a), (q,b), (r,p), (s,d)}
In[B3] = {(p,a), (q,b), (r,p), (s,d)}
Out[B3] = {(p,a), (q,a), (r,s), (s,d)}
In[B4] = {(p,a), (q,a), (q,b), (r,p) (r,s), (s,d)}
Out[B4] = {(p,a), (p,d), (q,e), (r,p) (r,s), (s,d)}