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.
(b) Perform reachability analysis.
(c) Perform common subexpression elimination.

2. Consider the following three address code in a basic block.

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

(a) Perform copy propagation.
(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.
3. Consider the following flow graph. Perform constant propagation and folding using MFP.

4. Consider the following CFG.

(a) Perform liveliness analysis. Show the Live sets at each point of the CFG. Identify the dead code.
(b) Assume that the system has five registers. Construct the interference graph. Perform color assignment. Determine the register for each variable.
(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.
5. Consider the following attribute grammar.

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

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

(a) Determine the number of registers required in the machine code generated for the input statement.
(b) Determine the register allocation for the computation.
(c) Generate the machine code using four instructions:

- \(\text{load register id/num : register = id/num}\)
- \(\text{add register1 register2 register3/id/num : register1 = register2 + register3/id/num}\)
- \(\text{mul register1 register2 register3/id/num : register1 = register2 * register3/id/num}\)
- \(\text{store register id : id = register}\)

6. Consider the following three address code.

\[
\begin{align*}
t1 &:= j \\
t1 &:= t1 * \text{dim2} \\
t1 &:= t1 + k \\
t1 &:= t1 * \text{w} \\
t2 &:= j * k \\
t3 &:= j \\
t3 &:= t3 * \text{dim2} \\
t3 &:= t3 + k \\
t3 &:= t3 * \text{w} \\
t4 &:= \text{B}[t3] \\
t5 &:= t2 + t4 \\
A[t1] &:= t5
\end{align*}
\]

(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.

- \(\text{load register id/num : register = id/num}\)
- \(\text{store register id : id = register}\)
- \(\text{loadarray register1 id register2 : register1 = id[register2]}\)
- \(\text{storearray register1 id register2 : id[register2] = register1}\)
- \(\text{add register1 register2 register3/id/num : register1 = register2 + register3/id/num}\)
- \(\text{mul register1 register2 register3/id/num : register1 = register2 * register3/id/num}\)
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>load &lt;reg&gt; &lt;id&gt;</td>
<td>load identifier &lt;id&gt; from memory into register &lt;reg&gt;</td>
<td>10</td>
</tr>
<tr>
<td>store &lt;reg&gt; &lt;id&gt;</td>
<td>store &lt;reg&gt; value into &lt;id&gt;’s memory location</td>
<td>10</td>
</tr>
<tr>
<td>aload &lt;reg1&gt; &lt;reg2&gt; &lt;reg3&gt;</td>
<td>load the content of array A[reg3] into register &lt;reg1&gt;, where reg2 = &amp;A (A’s base address)</td>
<td>11</td>
</tr>
<tr>
<td>astore &lt;reg1&gt; &lt;reg2&gt; &lt;reg3&gt;</td>
<td>store the content of register &lt;reg1&gt; into array A[reg3] into, where reg2 = &amp;A</td>
<td>15</td>
</tr>
<tr>
<td>add &lt;reg1&gt; &lt;reg2&gt; &lt;reg3&gt;</td>
<td>reg1 := reg2 + reg3</td>
<td>2</td>
</tr>
<tr>
<td>mul &lt;reg1&gt; &lt;reg2&gt; &lt;reg3&gt;</td>
<td>reg1 := reg2 * reg3</td>
<td>5</td>
</tr>
<tr>
<td>addc &lt;reg1&gt; &lt;reg2&gt; &lt;const&gt;</td>
<td>reg1 := reg2 + constant</td>
<td>2</td>
</tr>
<tr>
<td>mulc &lt;reg1&gt; &lt;reg2&gt; &lt;const&gt;</td>
<td>reg1 := reg2 * constant</td>
<td>5</td>
</tr>
<tr>
<td>addx &lt;reg1&gt; &lt;reg2&gt; &lt;id&gt;</td>
<td>reg1 := reg2 + id</td>
<td>11</td>
</tr>
<tr>
<td>mulx &lt;reg1&gt; &lt;reg2&gt; &lt;id&gt;</td>
<td>reg1 := reg2 * id</td>
<td>16</td>
</tr>
</tbody>
</table>

Consider the following three address code in a basic block.

\[
\begin{align*}
    t1 &:= j + 1 \\
    t1 &:= t1 * d2 \\
    t1 &:= t1 + k \\
    t1 &:= t1 * w \\
    t7 &:= B[t1] \\
    t2 &:= j * k \\
    t2 &:= t2 * w \\
    t8 &:= A[t2] \\
    t9 &:= t7 + t8 \\
    A[t1] &:= t9
\end{align*}
\]

(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.

(d) Tile the tree using the dynamic programming algorithm.

8. Consider the following program.

```plaintext```
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.
9. Consider the following flow graph.

(a) Identify and mark define-use links within the loop based on reachability analysis.
(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.
(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.

10. Consider the following flow graph. Perform strength reduction for the statements in the loop.
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.

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

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.