Consider the program:

```c
int c = x-y;
while (c > 0) {
    int x = 3*c;
    if (y < 0) y = -y*(a+b); else y = -4*(a+b);
    c = c-1;
}
int c = (a+b)*(x-y);
```

Its control flow graph, with node labels as superscripts, is:

![Control Flow Graph]

a) Provide the set $E$ of all nontrivial arithmetic subexpressions\(^1\).

b) For each node of the control flow graph, determine the *kill* and *gen* sets with respect to $E$.

c) For each node give the final set of very busy expressions at the entry and the exit of this node as generated by the *kill/gen* analysis (Note that you do not have to write all iteration steps). Additionally, provide optimizations for the given program according to the results of the very busy expressions analysis.

--

\(^1\)That is the set of all subtrees of the syntax tree of an arithmetical expression excluding those with only one node. Additionally, boolean expressions are also not taken into account.
2 Copy Analysis 10 Points

The copy analysis determines for each program point whether on every execution path leading to it from a copy assignment, say \( x := y \), there are no assignments to \( y \).

a) Argue whether the analysis should be a backward or a forward analysis.

b) Define the killgen sets for all possible statements.

c) Explain your analysis by means of a simple example.

3 Code Optimization 10 Points

Consider the following program:

```c
int a;

int multiply (int i, int n, int value) {
    if (i == n) return value+2*n;
    else return multiply(i+1, n, value+2*n);
}

int main (void) {
    a = readi ();
    a = multiply (1, a, 0);
    printi (a);
    return a;
}
```

The `code` function generates the following CMa code:

```
1 enter 6
2 alloc 1
3 mark
4 loadc .main
5 call 0
6 halt
7 .multiply enter 9
8 loadrc 1
9 load
10 loadc 2
11 load
12 eq
13 jump .1label
14 loadrc 3
15 load
16 loadc 2
17 loadrc 2
18 load
19 mul
20 add
21 storer -3
```

```
22 return
23 jump .2label
24 .1label mark
25 loadrc 1
26 load
27 loadc 1
28 add
29 loadrc 2
30 load
31 loadc 3
32 load
33 loadc 2
34 loadrc 2
35 load
36 mul
37 add
38 loadc .multiply
39 call 3
40 storer -3
41 return
42 .2label return
43 .main enter 8
44 readi
45 loadc 1
46 store
47 pop
48 mark
49 loadc 1
50 loadc 1
51 load
52 loadc 0
53 loadc .multiply
54 call 3
55 loadc 1
56 store
57 pop
58 loadc 1
59 load
60 print
61 loadc 1
62 load
63 storer -3
64 return
```
Describe seven optimizations that would improve the generated code (in terms of code size or required memory). The optimizations should not be specific to the above program and are thus supposed to be able to improve arbitrary input programs. For each of these optimizations:

1) Explain informally how it works.

2) Argue whether the optimization may be a postpass optimization (only operating on the CMa assembly) or needs access to (a representation of) the C code.

Explain which of these optimizations could be applied to the given program, argue whether there might be some conflicting optimizations, and give the assembly code after having applied the best optimizations.