1 Very Busy Expressions

a) \( E = \{ -y, c - 1, 3 \ast c, a + b, x - y, -4 \ast (a + b), -y \ast (a + b), (a + b) \ast (x - y) \}\)

b) kill and gen sets for each node:

\[
\begin{align*}
\text{kill}(1) &= \{ c - 1, 3 \ast c \} \\
\text{kill}(2) &= \emptyset \\
\text{kill}(3) &= \{ x - y, (a + b) \ast (x - y) \} \\
\text{kill}(4) &= \emptyset \\
\text{kill}(5) &= \{ -y, x - y, -y \ast (a + b), (a + b) \ast (x - y) \} \\
\text{kill}(6) &= \emptyset \\
\text{kill}(7) &= \{ c - 1, 3 \ast c \} \\
\text{kill}(8) &= \{ c - 1, 3 \ast c \} \\
\text{gen}(1) &= \{ x - y \} \\
\text{gen}(2) &= \emptyset \\
\text{gen}(3) &= \{ 3 \ast c \} \\
\text{gen}(4) &= \emptyset \\
\text{gen}(5) &= \{ -y, a + b, -y \ast (a + b) \} \\
\text{gen}(6) &= \{ a + b, -4 \ast (a + b) \} \\
\text{gen}(7) &= \{ c - 1 \} \\
\text{gen}(8) &= \{ a + b, x - y, (a + b) \ast (x - y) \}
\end{align*}
\]

c) Equations:

\[
\begin{align*}
\text{node} & | \text{\( V_{exit}(\text{node}) \)} & \text{\( V_{entry}(\text{node}) \)} \\
1 & (\text{\( V_{exit}(1) \setminus \text{\( \text{kill}(1) \)} \)) \cup \text{\( \text{gen}(1) \)} & \text{\( V_{entry}(2) \)} \\
2 & (\text{\( V_{exit}(2) \setminus \text{\( \text{kill}(2) \)} \)) \cup \text{\( \text{gen}(2) \)} & \text{\( V_{entry}(3) \cap \text{\( V_{entry}(8) \)} \)} \\
3 & (\text{\( V_{exit}(3) \setminus \text{\( \text{kill}(3) \)} \)) \cup \text{\( \text{gen}(3) \)} & \text{\( V_{entry}(4) \)} \\
4 & (\text{\( V_{exit}(4) \setminus \text{\( \text{kill}(4) \)} \)) \cup \text{\( \text{gen}(4) \)} & \text{\( V_{entry}(5) \cap \text{\( V_{entry}(6) \)} \)} \\
5 & (\text{\( V_{exit}(5) \setminus \text{\( \text{kill}(5) \)} \)) \cup \text{\( \text{gen}(5) \)} & \text{\( V_{entry}(7) \)} \\
6 & (\text{\( V_{exit}(6) \setminus \text{\( \text{kill}(6) \)} \)) \cup \text{\( \text{gen}(6) \)} & \text{\( V_{entry}(7) \)} \\
7 & (\text{\( V_{exit}(7) \setminus \text{\( \text{kill}(7) \)} \)) \cup \text{\( \text{gen}(7) \)} & \text{\( V_{entry}(2) \)} \\
8 & \text{\( \text{\( \text{gen}(8) \)} \setminus \text{\( \text{\( \text{kill}(7) \)} \)) \cup \text{\( \text{gen}(7) \)} & \emptyset
\end{align*}
\]
Largest solution:

<table>
<thead>
<tr>
<th>node</th>
<th>$V_{B_{\text{entry}}}(\text{node})$</th>
<th>$V_{B_{\text{exit}}}(\text{node})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${a+b, x-y}$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>2</td>
<td>${a+b}$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>3</td>
<td>${c-1, 3*c, a+b}$</td>
<td>${c-1, a+b}$</td>
</tr>
<tr>
<td>4</td>
<td>${c-1, a+b}$</td>
<td>${c-1, a+b}$</td>
</tr>
<tr>
<td>5</td>
<td>${-y, c-1, a+b, -y*(a+b)}$</td>
<td>${c-1, a+b}$</td>
</tr>
<tr>
<td>6</td>
<td>${c-1, a+b, -4*(a+b)}$</td>
<td>${c-1, a+b}$</td>
</tr>
<tr>
<td>7</td>
<td>${c-1, a+b}$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>8</td>
<td>${a+b, x-y, (a+b)*(x-y)}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Based upon the result of the very busy expressions analysis, the given program could be optimized as follows:

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

2 Copy Propagation

The copy analysis computes sets of pairs of variables. Such a pair $(x, y)$ at a program point $p$ indicates that there was an assignment $x := y$ on each path leading to this program point and no intermediate assignment to $x$ or $y$.

a) The analysis is obviously a forward analysis, since the data flow value of a program point depends on the path leading to it (i.e. on the prior copy assignment).

b) For the analysis only assignments are important, for all other statements the kill/gen-sets are empty and the data flow values are simply forwarded. The kill-set of an assignment consists of all pairs where the assigned variables occurs - no matter on which side of the pair. The gen-set of a copy assignment simply consists of the corresponding pair of assigned variables and the gen-set of an arbitrary assignment, but not an copy-assignment, is empty.

```latex
\begin{align*}
\text{kill}_{CA}(x := \_.) & = \{(x, \_.)\} \cup \{\_, x\} \\
\text{kill}_{CA}(\_.) & = \emptyset \\
\text{gen}_{CA}(x := y) & = \{(x, y)\} \\
\text{gen}_{CA}(x := \_.) & = \emptyset \\
\end{align*}
```

\[ CA_{\text{entry}}(l) = \begin{cases} \emptyset & \text{if } l = \text{init}(S) \\ \bigcap \{CA_{\text{exit}}(l')(l', l) \in \text{flow}(S)\} & \text{otherwise} \end{cases} \]

\[ CA_{\text{exit}}(l) = (CA_{\text{entry}}(l) \setminus \text{kill}_{CA}(B^l)) \cup \text{gen}_{CA}(B^l), \text{ where } B^l \in \text{blocks}(S). \]
c) Example:

\[\begin{align*}
  y & := z; \\
  a & := b; \\
  a & \neq z; \\
  b & := a + 5; \\
  a & := x; \\
  a & := z;
\end{align*}\]

<table>
<thead>
<tr>
<th>node</th>
<th>(VB_{\text{entry}}(\text{node}))</th>
<th>(VB_{\text{exit}}(\text{node}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>({y, z})</td>
</tr>
<tr>
<td>2</td>
<td>({(y, z)})</td>
<td>({(y, z), (a, b)})</td>
</tr>
<tr>
<td>3</td>
<td>({(y, z), (a, b)})</td>
<td>({(y, z), (a, b)})</td>
</tr>
<tr>
<td>4</td>
<td>({(y, z), (a, b)})</td>
<td>({(y, z)})</td>
</tr>
<tr>
<td>5</td>
<td>({(y, z), (a, b)})</td>
<td>({(y, z), (a, x)})</td>
</tr>
<tr>
<td>6</td>
<td>({(y, z)})</td>
<td>({(y, z), (a, z)})</td>
</tr>
</tbody>
</table>

3 Code Optimization

Amongst others, a compiler might consider applying the following optimizations:

1. Constant folding/propagation

   1) Identifies and propagates constants and evaluates constant expressions.
   2) This kind of optimization may be a postpass optimization and does not necessarily need access to the C code.

2. Dead-code elimination

   1) Removes unreachable code, such as functions that are never being called.
   2) The optimization operates on the syntax tree.

3. Strength reduction

   1) Identifies expensive computations depending on induction variables and tries to replace these by cheaper instruction sequences.
   2) Needs a representation of the C code.
4. Copy propagation
   1) Determines and replaces exact copies of variables. For instance, the following piece of code:

   \[
   x = y; \\
   z = 4\times x;
   \]

   would be transformed into:

   \[
   z = 4\times y;
   \]

   2) The optimization needs access to the C code.

5. Function inlining
   1) To safe function calls to some routine, the function body is inlined at the corresponding call sites. However, function inlining should only be applied to small routines. Otherwise, the increase in code size is probably not worth the effort.

   2) The optimization needs access to the C code.

6. Code selection
   1) Replaces sequences of instructions by more specialized, cheaper ones. For instance:

   ```
   loadrc 4 \\
   load
   ```

   would be transformed into:

   ```
   loadr 4
   ```

   2) The optimization operates on the assembly level.

7. Tail recursion
   1) The optimization removes recursive functions by replacing them with loops.

   2) Access to a representation of the C code is obviously required.

8. Loop unrolling
   1) Corresponds to function inlining. Small loops, where the number loop iterations is known, are inlined in the program code.

   2) This analysis needs access to the syntax tree.

9. Common subexpression elimination
   1) Determines subexpressions used throughout several expressions and stores the result of subexpressions commonly used in temporary variables.

   2) This analysis needs access to the syntax tree.
The following optimizations can be applied to the given program:

1. Tail recursion
2. Common subexpression elimination
3. Function inlining
4. Code selection

On the syntax level, the optimization would transform the input program into the following:

```c
int a;

int main (void)
{
    int i, t1, value;
    a = readi ();
    value = 0;
    t1 = 2*a;
    i = 1;
    for (; i <= a; ++i)
        value += t1;
    a = value;
    printi (a);
    return a;
}
```

The following CMa code would be generated:

```
1    enter 6
2    alloc 1
3    mark
4    loadc .main
5    call 0
6    halt
7    .main enter 5
8    alloc 3
9    readi
10   storea 1
11   pop
12   loadc 2
13   mula 1
14   storer 2
15   pop
16   loadc 1
17   storer 1
18    pop
19    .3label loadr 1
20    loada 1
21    leq
22    jumpz .1label
23    loadr 3
24    addr 2
25    storer 3
26    pop
27    .2label incr 1
28    jump .3label
29    .1label loadr 3
30    storea 1
31    dup
32    print
33    storer -3
34    return
```