Problem 1

\[
\begin{align*}
m &:= 0 \quad \text{LEADER} \\
i &:= 0 \\
L1: \quad &\text{if } i<n \Rightarrow \text{false goto } L2 \quad \text{LEADER} \\
s &:= 0 \quad \text{LEADER} \\
j &:= i \\
L3: \quad &\text{if } j<n \Rightarrow \text{false goto } L4 \quad \text{LEADER} \\
x &:= j \quad \text{LEADER} \\
s &:= s+x \\
\text{if } s>m \Rightarrow \text{false goto } L5 \\
m &:= s \quad \text{LEADER} \\
L5: \quad &\text{// end of if-statement} \\
j &:= j+1 \quad \text{LEADER} \\
goto L3 \\
L4: \quad &\text{// end of inner for-loop} \\
i &:= i+1 \quad \text{LEADER} \\
goto L1 \\
L2: \quad &\text{// end of outer for-loop} \\
\text{return } m \quad \text{LEADER}
\end{align*}
\]

The reason some labels are on lines of their own is to emphasize the fact that they are not generated at the same time as the code that follows them. Think about how an IR code generator would work. Given a for-statement, it would first emit code for the initializer (\(i:=0\)). Then, it would generate a label (L1:) and emit the condition (if \(i<n\Rightarrow\text{false}\)). If the condition is false, control will continue at some other point, so it should generate a label for that destination and emit a goto statement (goto L2). Then, go on to generate code for the body of the loop. At the end of that, place the destination label (L2:). What comes next is the code for the statement following the loop and will be generated separately. See the SDT for the while-loop in problem 2 for more info. This is very similar to what you will have to do in PA4.

The first step in the SSA conversion is to find out in which blocks each variable is defined and insert a \(\phi\)-function in the dominance frontiers of those blocks. The \(\phi\)-functions are also definitions of the variable.

This information is shown in figure 2.
Figure 1: The CFG and dominator tree. The nodes in the dominator tree are labeled with their dominance frontiers.

<table>
<thead>
<tr>
<th>Variable</th>
<th>BB where it is defined</th>
<th>BB with $\phi$–functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>B1, B7, B8, B4, B2</td>
<td>B8, B4, B2</td>
</tr>
<tr>
<td>$i$</td>
<td>B1, B5, B2</td>
<td>B2</td>
</tr>
<tr>
<td>$s$</td>
<td>B3, B2, B6, B4</td>
<td>B2, B4</td>
</tr>
<tr>
<td>$j$</td>
<td>B3, B2, B8, B4</td>
<td>B2, B4</td>
</tr>
<tr>
<td>$x$</td>
<td>B6, B4, B2</td>
<td>B4, B2</td>
</tr>
</tbody>
</table>

Figure 2: Finding where $\phi$–functions should be placed
After inserting the $\phi$—functions, we need to rename. The final result is shown in figure 3.

Note how each argument $i$ of a $\phi$—function is taken from the corresponding predecessor $i$ as ordered in the figure. Furthermore, $n$ doesn’t have an initial definition, so we use the pound symbol as a subscript.

The conversion from SSA back to MIR is shown in figure 3. $m_4 := m_2$ provides an interesting challenge. Technically, it should be at the end of the block, right after the evaluation of $s_3 > m_2$ and before the `goto`. Nevertheless, its current location is safe. A better representation would have created a temporary variable to hold the result of the test and then used that variable in a branching statement.
Problem 2

In the SDT that follows, \( \| \) stands for “concatenate” and \textit{newTemp()}\ is a function that generates a new temporary variable every time it is called.

The SDT follows.

\[
\begin{align*}
< S > & \rightarrow \text{while } < E > \text{ do } \{ < S_1 > \} \\
& \{ \\
& S.\text{begin} = \text{genLabel()} \\
& S.\text{end} = \text{genLabel()} \\
& S.\text{code} = S.\text{begin} \quad \| \quad “:\" \\
& \| E.\text{code} \\
& \| “ \text{if }” \| E.\text{temp} \| “== \text{FALSE goto }” \| S.\text{end} \\
& \| S_1.\text{code} \\
& \| “ \text{goto }” \| S.\text{begin} \\
& \| S.\text{end} \| “:\" \\
& \}
\end{align*}
\]

\[
\begin{align*}
< S > & \rightarrow \text{if } < E > \text{ then } \{ < S_1 > \} \\
& \{ \\
& S.\text{end} = \text{genLabel()} \\
& S.\text{code} = E.\text{code} \\
& \| “ \text{if }” \| E.\text{temp} \| “== \text{FALSE goto }” \| S.\text{end} \\
& \| S_1.\text{code} \\
& \| S.\text{end} \| “:\" \\
& \}
\end{align*}
\]

\[
\begin{align*}
< S > & \rightarrow < S_1 > ; < S_2 > \\
& \{ \\
& S.\text{code} = S_1.\text{code} \| S_2.\text{code} \\
& \}
\end{align*}
\]

\[
\begin{align*}
< S > & \rightarrow < V > := < E > \\
& \{ \\
& S.\text{code} = E.\text{code} \\
& \| V.\text{name} \| “:=” \| E.\text{temp} \\
& \}
\end{align*}
\]

\[
\begin{align*}
< E > & \rightarrow < E_1 > == < E_2 > \\
& \{ \\
& E.\text{temp} = \text{newTemp()} \\
& E.\text{code} = E_1.\text{code} \| E_2.\text{code} \\
& \quad E.\text{temp} \| “:=” \| E_1.\text{temp} \quad “==” \quad E_2.\text{temp} \\
& \}
\end{align*}
\]
\[
\begin{align*}
\langle E \rangle &\quad\rightarrow\quad \langle E_1 \rangle + \langle E_2 \rangle \\
\{ &\quad E.\text{temp} = \text{newTemp()} \\
&\quad E.\text{code} = E_1.\text{code} \ || \ E_2.\text{code} \\
&\quad \quad \quad \quad \quad \quad E.\text{temp} \ || \ "=" || E_1.\text{temp} \ "+" E_2.\text{temp} \\
\}
\end{align*}
\]

\[
\begin{align*}
\langle E \rangle &\quad\rightarrow\quad \text{id} \\
\{ &\quad E.\text{temp} = \text{id.name} \\
&\quad E.\text{code} = "" \\
\}
\end{align*}
\]

Now let us consider the addition of a break statement and how this will affect the code. A sample syntax tree for a possible nested-loop configuration is shown in figure 4.

![Figure 4: The blue arrows show how the value of attribute $S_1.\text{end}$ needs to travel in the parse tree](image)

The first break statement ($S_5$) should generate a branch just outside loop $S_3$. The label used in this goto statement is $S_3.\text{end}$. Things become more interesting when we consider the other break ($S_6$). This should generate a \textbf{goto $S_1.\text{end}$}. The label $S_1.\text{end}$ was created when we started generating code for $S_1$, yet now we need it in $S_6$. This piece of information needs to pass from $S_1$ to $S_2$ to $S_4$ to $S_6$. Clearly, it’s an inherited attribute that will have to be passed along all kinds of statements that may consist of other statements (eg. $S \rightarrow S \ S$).

Below are the rules we need to add to the existing SDT:
< S >  →  while < E > do { < S₁ > }
{  
    S₁.in  =  S.end
}

< S >  →  if < E > then { < S₁ > }
{  
    S₁.in  =  S.in
}

< S >  →  < S₁ > ; < S₂ >
{  
    S₁.in  =  S.in
    S₂.in  =  S.in
}

< S >  →  break
{  
    S.code  =  "goto" || S.in
}