Exercise 1: Trace Abstraction  
In this task, you should apply trace abstraction to prove that a program, here given by its control-flow graph, is safe.

Consider the following control-flow graph for a program $P$, and let $A_P$ be the corresponding automaton.

![Control-flow graph](image)

Give two error traces $\pi_1$ and $\pi_2$ and construct corresponding Floyd-Hoare automata $A_1$ and $A_2$ such that the inclusion $L(A_P) \subseteq L(A_1) \cup L(A_2)$ holds.

Exercise 2: Termination  
In the lecture, we discussed four different properties of programs. One property was termination the other properties were related to termination. We provide formal definitions here. In each case, we consider a program $P$ with a CFG $(\text{Loc}, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$.

(a) We say that $P$ can reach the exit location if there exists a finite execution, such that the first configuration $(\ell, s)$ is initial, and the last configuration is $(\ell_{\text{ex}}, s')$ for some state $s'$.

(b) We say that $P$ can stop if there exists a reachable configuration $(\ell, s)$ such that there exists no configuration $(\ell', s')$ and statement $st$ with $(\ell, st, \ell') \in \Delta$ and $(s, s') \in [st]$. 
(c) We say that $P$ always reaches the exit location if there exist no infinite executions, and all finite executions end in a configuration $(\ell', s')$ where we either have a successor (i.e., there exists a configuration $(\ell'', s'')$ and statement $st$ with $(\ell', st, \ell'') \in \Delta$ and $(s', s'') \in [st]$) or we have that $\ell'$ is $\ell_{ex}$.

(d) We say that $P$ always stops (resp. $P$ terminates) if there exist no infinite executions.

In this exercise, you should give programs that differentiate between these definitions. In particular, for each of the following pairs, give a program such that one definition holds but the other does not. Explain which of the definitions holds and why.

(a) $P$ can reach the exit location vs. $P$ can stop

(b) $P$ can stop vs. $P$ always stops

Exercise 3: Ranking Functions

For each of the following programs, state whether it (always) terminates or not. If it terminates, give a ranking function for each loop in the program. If it may not terminate, give an infinite execution of the program.

```plaintext
while (x > 0) {
  while (y > 0) {
    y := y - 1;
  } else {
    x := x - 1;
    havoc y;
  }
}
```

Listing 1: Program $P_1$

```plaintext
while (x > 0) {
  if (y > 0) {
    y := y - 1;
  } else {
    x := x - 1;
    havoc y;
  }
}
```

Listing 2: Program $P_2$

```plaintext
while (x > 0) {
  if (y > 0) {
    y := y - 1;
  } else {
    x := x - 1;
    havoc y;
  }
}
```

Listing 3: Program $P_3$

**Hint:** For simple loops it is often convenient to use a function whose range is $\mathbb{N}$ and the strictly greater than relation $>$ on natural numbers. For more complex loops, this is sometimes not sufficient but we can use instead a function $f : S_{\nu, \mu} \rightarrow \mathbb{N} \times \ldots \times \mathbb{N}$ whose range are $n$-tuples of natural numbers and the lexicographic order $>_{\text{lex}}$ that we define as follows.

$$(m_1, \ldots, m_n) >_{\text{lex}} (m'_1, \ldots, m'_n) \text{ iff there exists } i \in \{1, \ldots, n\} \text{ such that } m_i > m'_i$$

and for all $k \in \{1, \ldots, i - 1\}$ the equality $m_k = m'_k$ holds.

If a function with that signature together with the order $>_{\text{lex}}$ is a ranking function, it is often called a lexicographic ranking function.