1 Axiomatic semantics (Hoare logic)

Hoare logic provides a set of rules for proving the correctness of computer programs. It deals with sentences of the form \( \{p\} A \{q\} \), where \( p \) and \( q \) are assertions (i.e. they can be either true or false), and \( A \) is a program. The meaning is that if the precondition \( p \) is true, and we run the program \( A \), then the postcondition \( q \) is true. All the possible programs are defined by the following five rules:

\[
\begin{align*}
\{ q[E/X] \} & \quad X := E \{ q \} \quad \text{(R1)} \\
p \land \neg c = q & \quad \{ p \land c \} \ A \{ q \} \quad \{ p \} \text{ if } c \text{ then } \ A \{ q \} \quad \text{(R2)} \\
\{ p \} \ A \{ r \} & \quad \{ r \} \ B \{ q \} \quad \{ p \} \ A; B \{ q \} \quad \text{(R3)} \\
\{ p \land c \} \ A \{ p \} & \quad \{ p \} \text{ while } c \text{ do } \ A \{ \neg c \land p \} \quad \text{(R4)} \\
p \Rightarrow p' & \quad \{ p' \} \ A \{ q' \} \quad q \Rightarrow q' \quad \{ p \} \ A \{ q \} \quad \text{(R5)}
\end{align*}
\]

Rule R4 is the hardest to interpret. To use it, one has to find the loop invariant \( p \) that is true after each iteration of the loop. Rule R5 allows to simplify assertions using mathematics.
1.1 Example: Factorial

We want to prove the correctness of the program

\[
\{ n \geq 0 \} \\
x = 1; \\
y = 1; \\
\text{while } x < n \text{ do} \\
x = x + 1; \\
y = y \cdot x; \\
\text{end; } \\
z = y; \\
\{ z = n! \}.
\]

To do so, we first use the composition rule R3 to break down the program into simpler ones by finding a postcondition for each command that is also a precondition for the next command (up to implications using R5). To find a valid precondition for the loop, we need to identify the loop invariant. In this case it is

\[
\text{\cdot} \equiv y = x! \land (x \leq n \lor (n = 0 \land y = 1)).
\]

Once we figure out all the assertions, we can check that they prove the correctness using the following control flow diagram:

Identifying the loop invariant is the hardest part of the proof and there is no general recipe for finding it. The control flow diagram has the advantage that it is easy to read however the correctness of the program can also be proven using standard notation for derivations as follows:

\[
\begin{align*}
R1_{\{n \geq 0\}} & x = 1 \ (n \geq 0 \land x = 1) \\
R3 & \text{while } x < n \text{ do } x = x + 1; \\
R4 & y = y \cdot x; \\
& \text{end; } z = y \ (z = n!) \\
R3 & \{ n \geq 0 \} \ x = 1 \ (n \geq 0 \land x = 1) \\
R1_{\{n \geq 0\}} & y = 1 \ (n \geq 0 \land y = 1) \\
R3 & \text{while } x < n \text{ do } x = x + 1; \\
R4 & y = y \cdot x; \\
& \text{end; } z = y \ (z = n!)
\end{align*}
\]
where

\[ \Phi_1 \equiv y = (x - 1)! \land (x \leq n \lor (n = 0 \land y = 1)); \]

## 2 Denotational semantics

A program is a map (function) from the set of states (\( \Sigma \)) to itself,

\[ [A] : \Sigma \to \Sigma. \]  

(1)

How should we represent functions? The notation \( f(x) = 2x + 1 \) is not well suited for our purposes since it says that \( f(x) \) is an expression. A possible solution is the so called \( \lambda \)-notation, which can be used to represent \( f \) as \( \lambda x. (2x + 1) \). In general, \( f = \lambda s.t \) means take the input \( s \) and substitute it into the expression \( t \).

We a different approach that represents programs as function from state to state. A state is represented as a list of tuples that associate every program variable with a value, for example \( \{(x, 3), (y, 0), (z, 0)\} \). We have the following semantics for our programs:

\[ [x = c] = \lambda s. \{(x, [c](s))\} \cup \{(y, s(y))|y \neq x\}, \]

\[ [A; B] = \lambda s. [B]([A](s)), \]

\[ [\text{if } c \text{ then } A] = \lambda s. \begin{cases} [A](s) & \text{if } [c](s) = \text{true} \\ [B](s) & \text{if } [c](s) = \text{false} \end{cases} \]

\[ [\text{while } c \text{ do } A] = \lambda s. \begin{cases} [\text{while } c \text{ do } A](s) & \text{if } [c](s) = \text{true} \\ [A](s) & \text{if } [c](s) = \text{false} \end{cases} \]

We see that the last equation defines the while command as a fixed point of some function. More precisely, we want it to be the least fixed point (least in some lattice).

### 2.1 Example: fixed points

Usually, in our programs the fixed point is unique:

\begin{verbatim}
while x < 3 do x = x + 1
\end{verbatim}

\((x, 3) \to (x, 3); (x, 4) \to (x, 4)\)

but \((x, 2) \to (x, 3); (x, 1) \to (x, 3); (x, 0) \to (x, 3)\)

However, in some programs there can be infinitely many fixed points because the while loop does not terminate:

\begin{verbatim}
while x > 2 do x = x + 1
\end{verbatim}

3
(x, 0) → (x, 0); (x, 1) → (x, 1); (x, 2) → (x, 2); (x, 3) → (x, 17); (x, 4) → (x, 17); (x, 3) → (x, 27); (x, 4) → (x, 27); ... leading to nontermination ("⊥" here stands for "undefined"): (x, 3) → (x, ⊥); (x, 4) → (x, ⊥)