The first proof of a complete program that we will show is the algorithm for computing the multiplication of two numbers by repeated addition. The program code is:

```plaintext
p := 0;
n := x;
while n greater 0 do
    p := p plus y;
n := n minus 1
end
```

Here, x and y are assumed to have initial values (we could add them to the initial store in operational semantics). We will prove that this program is correct relative to the specification \([ x \geq 0, p = x \times y ]\). Firstly, notice that if x is negative, the loop will terminate immediately, but p will have the wrong value. However, if x is zero, then, even though the loop terminates immediately, it still produces the right result (i.e. zero). If \( P \) is the whole program, we will try to prove:

\[
\{ x \geq 0 \} P \{ p = x \times y \}
\]

**PROOF STRATEGY**

Any program with a loop will need a proof in at least s many parts as there are loops. Each loop needs the following procedure:

1. Find a suitable invariant for the loop; this is an assertion that is true before the loop, while the loop continues, and after it terminates. Call this I.
2. Prove \( \{ I \land b \} \{ C \} \), where b is the arithmetic expression corresponding to the loop test, and C is the loop body.
3. Assert \( \{ I \} \) while B do C end \( \{ I \land \neg b \} \)

To put the other non-loop commands into the proof, work backwards through assignments and/or conditionals until the pre-condition. The proof is completed by showing that (if necessary) the post-condition for the loop implies the post-condition for the whole program (it could be equivalent) and possibly show that the initial condition implies the pre-condition for the first command, whatever it is. There may also be occasion to “glue” sections together by proving the appropriate implication and using either the strengthening or the weakening axiom.

**FINDING THE INVARIANT**

There is no known mechanical way to find the right invariant; some guesswork and/or creativity is needed. However, there are a few tricks that can be used. Whatever invariant is chosen, it should satisfy some basic requirements:

1. It should involve all assigned variables (i.e. variables that change program state)
2. It should always involve the loop exit test or something like it.
3. It should be as simple, but as strong as possible.

There are invariants that will not help the proof, so any invariant will not do; there are invariants that try to say too much, and are hard to work into the proof without error. Somewhere in the middle is the appropriate invariant. Every loop has an infinite number of them, but only one (or at most a small handful) will allow the proof to progress. In our case, we can use an investigation of the relationships among the variables when the loop runs. We could write the following table for a simple concrete example. Let us take \( x = 3 \) and \( y = 4 \). The changes the loop body makes are then contained in the table:
There is a pattern which is simply captured by the expression \( p = (3 - n) \times 4 \). We could also express the table generically:

\[
\begin{array}{cc}
p & n \\
0 & 3 \\
4 & 2 \\
8 & 1 \\
12 & 0 \\
\end{array}
\]

where the pattern is even clearer: \( p = (x - n) \times y \). Is this the invariant? Well it is part of it. Notice that, since the loop is the last command we need to show that \( I \land \neg b \Rightarrow q \), where \( q \) is the final assertion, i.e. \( p = x \times y \). Clearly we need to show that \( n=0 \), and we will only be able to do that if we include the termination condition as part of the invariant. Later, we will show that this is important for proving that the loop terminates, in complete correctness. For the moment, though, we just include \( n \geq 0 \) because without it the proof is impossible. The correct invariant is thus:

\[
I \equiv p = (x - n) \times y \land n \geq 0
\]

Now we need to prove \( \{ I \land n > 0 \} p := p + y; n := n - 1 \). We do this by using the assignment axiom twice and then the sequence axiom:

\[
\begin{align*}
\{ p = (x - (n-1)) \times y \land (n-1) \geq 0 \} & n := n - 1 \{ p = (x - n) \times y \land n \geq 0 \}, \text{ and} \\
\{ p + y = (x - (n-1)) \times y \land (n-1) \geq 0 \} & p := p + y \{ p = (x - (n-1)) \times y \land (n-1) \geq 0 \}
\end{align*}
\]

Manipulating the pre-condition we get

\[
\begin{align*}
p + y = (x - (n-1)) \times y \land (n-1) \geq 0 \Rightarrow p = (x - n) \times y \land n \geq 0 \Rightarrow p = (x - n) \times y \land n > 0
\end{align*}
\]

Luckily, this is exactly \( I \land n > 0 \), so we can apply the loop rule to get:

\[\{ I \} \text{ while } \ldots \{ I \land \neg (n > 0) \}\]

This completes part one of the proof.

**COMPLETING THE PROOF**

First, we can complete the proof of the program post-condition, \( p = x \times y \). We have to show that

\[
I \land \neg (n > 0) \Rightarrow p = x \times y
\]

Since the invariant contains \( n \geq 0 \), we need to show that \( n \geq 0 \land \neg (n > 0) \Rightarrow n = 0 \), since then the computational part reduces to \( p = x \times y \). This clearly the case since \( n \) cannot be both greater and less than zero.

The rest is simply to push the invariant back through the initialization assignments to produce a weakest pre-condition for the whole program. Using the assignment axiom, we have:

\[
\begin{align*}
\{ p = (x - x) \times y \land x \geq 0 \} & n := x \{ p = (x - n) \times y \land n \geq 0 \}, \text{ and} \\
\{ 0 = (x - x) \times y \land x \geq 0 \} & p := 0 \{ p = (x - x) \times y \land x \geq 0 \}
\end{align*}
\]
The pre-condition clearly reduces to \( \{x \geq 0\} \), which is exactly what we wanted to show, so the proof is complete. We did not need to use any weakening or strengthening, since everything worked out exactly. This is not always the case.