

Another Loop Example

$P \equiv \text{while } x \neq 0 \text{ do } s := s + x ; x := x - 1 \text{ od}$

$\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow$

$$\frac{\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2}{\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma}$$

Another Loop Example

$P \equiv \text{while } x \neq 0 \text{ do } s := s + x ; x := x - 1 \text{ od}$

$\{s \mapsto -15, x \mapsto -7\}(x \neq 0) \Rightarrow \text{True} \quad \dots$

$$\frac{\{s \mapsto -9, x \mapsto -6\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -9, x \mapsto -6\}(s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -15, x \mapsto -7\} \quad \{s \mapsto -15, x \mapsto -7\}(P) \Rightarrow}{\{s \mapsto -9, x \mapsto -6\}(P) \Rightarrow}$$

$$\frac{\{s \mapsto -4, x \mapsto -5\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto -4, x \mapsto -5\}(s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\}}{\{s \mapsto -4, x \mapsto -5\}(P) \Rightarrow}$$

$$\frac{\{s \mapsto 0, x \mapsto -4\}(x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto 0, x \mapsto -4\}(s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\}}{\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow}$$

$\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow$

This is not a direct proof of non-termination!

$$\frac{\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2}{\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma}$$

Operational Semantics

- Useful for exploration
- Useful to guide **implementation**
- Useful to show correctness of implementation
- Derived assertions correspond to **individual test cases**
- More general statements need to be shown at the meta-level
- **Not useful** to prove **general properties** of programs
 - **termination**
 - **correctness**

Correctness

- **Correctness is always relative to a specification**
- A specification is — in general — a logical formula
 - many different logics are used!
- A program is **correct** iff it **satisfies** its specification
- Using logical methods to prove correctness is called **formal verification**
 - Using (normally human-aided) *syntactic* methods: **proving**
 - normally necessary for functional requirements
 - Using (exhaustive, automated) *semantic* methods: **model checking**
 - most useful for safety & liveness properties (finite models)
- **How do you show a specification is correct?**
 - **Validation:** Are we building the right product?
 - **Verification:** Are we building the product right?

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

$$\{P\}S\{Q\}$$

where P and Q are formulae denoting conditions on **execution states**:

- P is the **precondition**
- S is a program fragment (statement)
- Q is the **postcondition**

A Hoare triple $\{P\}S\{Q\}$ has two readings:

Total correctness: *If* S starts in a state satisfying P ,
 then it terminates and its terminating state satisfies Q
 — “ S is **totally correct with respect to** P and Q ”

Partial correctness: *If* S starts in a state satisfying P and *terminates*,
 then its terminating state satisfies Q
 — “ S is **partially correct with respect to** P and Q ”

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Therefore:

- Operational semantics facilitates investigation of examples (“*testing*”)
- Axiomatic semantics facilitates relating a program with its specification
 — **verification**

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s ”

For example: • $\{x \mapsto 5, y \mapsto 7\} \models x > 0$

- $\{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55$

- $\{x \mapsto 5, y \mapsto 7\} \not\models x > y$

Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “ $s \models P$ ” means “condition P **holds**, or **is valid**, in state s ”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

Partial correctness: *If* S starts in a state satisfying P and *terminates*,
 then its terminating state satisfies Q

I.e.: For all states σ_1 and σ_2 ,
 if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, then $\sigma_2 \models Q$

Total correctness: *If* S starts in a state satisfying P ,
 then it terminates and its terminating state satisfies Q

I.e.: For all states σ_1 , if $\sigma_1 \models P$,
 then there is a state σ_2
 such that $\sigma_1(S) \Rightarrow \sigma_2$, and $\sigma_2 \models Q$

Proving Partial and Total Correctness

Total correctness of $\{P\} S \{Q\}$

is equivalent to

partial correctness of $\{P\} S \{Q\}$ *together with* the fact that S terminates when started in a state satisfying P

⇒ usually, separate **termination proof!**

- For partial correctness, it is relatively easy to give a **direct proof calculus**
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — *well-ordered* sets)

Unless explicitly mentioned, we read “ $\{P\} S \{Q\}$ ” as meaning **partial correctness**.

Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:
$$\frac{P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

Sequence:
$$\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

Conditional:
$$\frac{\{P \wedge b\}S_1\{Q\} \quad \{P \wedge \neg b\}S_2\{Q\}}{\{P\}\mathbf{if } b \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ fi}\{Q\}}$$

while-Loop:
$$\frac{\{INV \wedge b\}S\{INV\}}{\{INV\}\mathbf{while } b \mathbf{ do } S \mathbf{ od}\{INV \wedge \neg b\}}$$

Axiom Schema for Assignments

$$\{P[x \setminus e]\}x := e\{P\}$$

Examples:

- $\{2 = 2\}x := 2\{x = 2\}$
- $\{x + 1 = 2\}x := x + 1\{x = 2\}$
- $\{n + 1 = 2\}x := n + 1\{x = 2\}$

Typically, Hoare triples are derived starting from the *postcondition* — **backward reasoning**.

Considering this axiom schema as a way to *calculate* a precondition from assignment and postcondition, it calculates the **weakest precondition** that completes a valid Hoare triple.

Example Verification

$$\{\text{True}\}k := 0; s := 0; \mathbf{while } k \neq n \mathbf{ do } k := k + 1; s := s + k \mathbf{ od}\{s = \sum_{i=1}^n i\}$$

Example Annotated Program

$$\{\text{True}\} \Rightarrow \{0 = \sum_{i=1}^0 i\}$$

$$k := 0; \quad \{0 = \sum_{i=1}^k i\}$$

$$s := 0; \quad \{s = \sum_{i=1}^k i\}$$

while $k \neq n$

do $\{s = \sum_{i=1}^k i \wedge k \neq n\} \Rightarrow \{s + k + 1 = \sum_{i=1}^{k+1} i\}$

$$k := k + 1; \quad \{s + k = \sum_{i=1}^k i\}$$

$$s := s + k \quad \{s = \sum_{i=1}^k i\}$$

od

$$\{s = \sum_{i=1}^k i \wedge k = n\} \Rightarrow \{s = \sum_{i=1}^n i\}$$

Example Verification

$$\{\text{True}\}k := 0; s := 0; \text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od} \{s = \sum_{i=1}^n i\}$$

$$\Leftarrow \{\text{True}\}k := 0; s := 0; \text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od} \{s = \sum_{i=1}^k i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0; s := 0 \{s = \sum_{i=1}^k i\}$$

$$\wedge \{s = \sum_{i=1}^k i\} \text{while } k \neq n \text{ do } k := k + 1; s := s + k \text{ od} \{s = \sum_{i=1}^k i \wedge k = n\}$$

$$\Leftarrow \{\text{True}\}k := 0 \{0 = \sum_{i=1}^k i\}$$

$$\wedge \{0 = \sum_{i=1}^k i\} s := 0 \{s = \sum_{i=1}^k i\}$$

$$\wedge \{s = \sum_{i=1}^k i \wedge k \neq n\} k := k + 1; s := s + k \{s = \sum_{i=1}^k i\}$$

Example Verification (ctd.)

$$\Leftarrow (\text{True} \Rightarrow 0 = \sum_{i=1}^0 i) \wedge \{0 = \sum_{i=1}^0 i\} k := 0 \{0 = \sum_{i=1}^k i\}$$

$$\wedge \text{True}$$

$$\wedge \{s = \sum_{i=1}^k i \wedge k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^k i\}$$

$$\wedge \{s + k = \sum_{i=1}^k i\} s := s + k \{s = \sum_{i=1}^k i\}$$

$$\Leftarrow \text{True}$$

$$\wedge s = \sum_{i=1}^k i \wedge k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i$$

$$\wedge \{s + k + 1 = \sum_{i=1}^{k+1} i\} k := k + 1 \{s + k = \sum_{i=1}^k i\}$$

$$\wedge \text{True}$$

$$\Leftarrow \text{True}$$

Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
Start to consider the postcondition and how the last statement achieves it
- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration
- For **while** loops, the postcondition needs to consist of
 - the **invariant** of this loop, and
 - the negation of the **loop** condition

Auxiliary variables used in a loop are usually involved in the invariant!

Given a loop “**while** b **do** S **od**” and a postcondition Q , use the consequence rule to strengthen Q to Q' , such that

- $Q' \Rightarrow Q$ (strengthening)
- Q' involves all auxiliary variables — **generalisation!**
- Q' is of shape $INV \wedge \neg b$

Simultaneous Assignments

$$\{P[x_1 \setminus e_1, \dots, x_n \setminus e_n]\}(x_1, \dots, x_n) := (e_1, \dots, e_n)\{P\}$$

Examples:

- $\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}$
- $\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}$

Simultaneous assignments

- shorten code
- save auxiliary variables (for example for swapping)
- make proofs easier
- **require simultaneous substitution**

Example Problems (with Simultaneous Assignments)

$$\{n \geq 0\} (y, a, b) := (0, 1, 1); \\ \text{while } y \neq n \text{ do } (y, a, b) := (y + 1, b, a + b) \text{ od } \{a = fib_n\}$$

Given an n -element C-like array s , prove partial correctness:

$$\{\text{True}\} \\ (i, a) := (0, 0); \\ \text{while } i \neq n \\ \text{do if } x = s[i] \\ \text{then } (i, a) := (i + 1, a + 1) \\ \text{fi od} \\ \{a = \#\{j : \mathbb{N} \mid s[j] = x \wedge 0 \leq j < n\}\}$$

What does this program do?

Fibonacci

$$\{n \geq 0\} (y, a, b) := (0, 1, 1); \\ \text{while } y \neq n \text{ do } (y, a, b) := (y + 1, b, a + b) \text{ od } \{a = fib_n\} \\ \Leftrightarrow \langle (\text{right consequence}) \rangle \\ \{n \geq 0\} P \{a = fib_y \wedge b = fib_{y+1} \wedge y = n\} \\ \wedge (a = fib_y \wedge b = fib_{y+1} \wedge y = n \Rightarrow a = fib_n) \\ \Leftrightarrow \langle (\text{sequence, logic}) \rangle \\ \{n \geq 0\} (y, a, b) := (0, 1, 1) \{a = fib_y \wedge b = fib_{y+1}\} \wedge \\ \{a = fib_y \wedge b = fib_{y+1}\} \text{while } y \neq n \text{ do } A \text{ od } \{a = fib_y \wedge b = fib_{y+1} \wedge y = n\} \\ \wedge \text{True}$$

Fibonacci (ctd.)

$$\Leftrightarrow \langle (\text{left consequence, while}) \rangle \\ (n \geq 0 \Rightarrow 1 = fib_0 \wedge 1 = fib_{0+1}) \\ \wedge \{1 = fib_0 \wedge 1 = fib_{0+1}\} (y, a, b) := (0, 1, 1) \{a = fib_y \wedge b = fib_{y+1}\} \\ \wedge \{a = fib_y \wedge b = fib_{y+1} \wedge y \neq n\} (y, a, b) := (y + 1, b, a + b) \\ \{a = fib_y \wedge b = fib_{y+1}\} \\ \Leftrightarrow \langle (\text{arithmetic, assignment, left consequence}) \rangle \\ \text{True} \wedge \text{True} \\ \wedge (a = fib_y \wedge b = fib_{y+1} \wedge y \neq n \Rightarrow b = fib_{y+1} \wedge a + b = fib_{(y+1)+1}) \\ \wedge \{b = fib_{y+1} \wedge a + b = fib_{(y+1)+1}\} (y, a, b) := (y + 1, b, a + b) \\ \{a = fib_y \wedge b = fib_{y+1}\} \\ \Leftrightarrow \langle (\text{arithmetic, assignment}) \rangle \\ \text{True} \wedge \text{True}$$

Array Traversal

Given an n -element C-like array s , prove partial correctness:

```

{True}
(i, a) := (0, 0);
while i ≠ n
do   if x = s[i]
     then (i, a) := (i + 1, a + 1)
fi od
{a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < n} }

```

```

{True} P {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < n} }
⇐ ⟨ ( right consequence ) ⟩
{True} Init ; W {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i = n}
  ∧ ((a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i = n) ⇒ Post)
⇐ ⟨ ( sequence , logic ) ⟩

```

```

{True} (i, a) := (0, 0) {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} }
  ∧ {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} } W
  {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i = n}
  ∧ True
⇐ ⟨ ( left consequence , while ) ⟩
(True ⇒ 0 = #{j : ℕ | s[j] = x ∧ 0 ≤ j < 0})
  ∧ {0 = #{j : ℕ | s[j] = x ∧ 0 ≤ j < 0} } (i, a) := (0, 0)
  {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} }
  ∧ {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i ≠ n} B
  {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} }
⇐ ⟨ ( logic and arithmetic , assignment , conditional ) ⟩
True ∧ True
  ∧ {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i ≠ n ∧ x = s[i]}
  (i, a) := (i + 1, a + 1) {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} }
  ∧ ((a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i ≠ n ∧ x ≠ s[i]) ⇒ a = #{j :
  ℕ | s[j] = x ∧ 0 ≤ j < i})

```

```

⇐ ⟨ ( left consequence , logic ) ⟩
((a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} ∧ i ≠ n ∧ x = s[i])
  ⇒ a + 1 = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i + 1}
  ∧ {a + 1 = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i + 1}} (i, a) := (i + 1, a + 1)
  {a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < i} }
  ∧ True
⇐ ⟨ ( logic , assignment ) ⟩
True ∧ True

```

Array Traversal

Given an n -element C-like array s , prove partial correctness:

```

{True}
(i, a) := (0, 0);
while i ≠ n
do   if x = s[i]
     then (i, a) := (i + 1, a + 1)
fi od
{a = #{j : ℕ | s[j] = x ∧ 0 ≤ j < n} }

```

What does this program do?

Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- **Different languages have different rules**
- Such rule sets are **specifications** of **language implementations**
- We **define the rules** for language features and extensions
- We **justify the rules** against different presentations of the defined features
- We **derive the rules** e.g. from source-to-source translations

repeat ... until ...

Operational Semantics:

$$\frac{\sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(b) \Rightarrow \text{True}}{\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_1}$$

$$\frac{\sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(b) \Rightarrow \text{False} \quad \sigma_1(\text{repeat } s \text{ until } b) \Rightarrow \sigma_2}{\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_2}$$

Axiomatic Semantics:

$$\frac{\{INV\} S \{INV\}}{\{INV\} \text{repeat } s \text{ until } b \{INV \wedge b\}}$$

Proper for-Loops — Operational Semantics

Proper for-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset

Operational Semantics:

$$\frac{\sigma(\text{beg}) \Rightarrow b \quad \sigma(\text{end}) \Rightarrow e \quad \sigma_b = \sigma \quad \forall i : \mathbf{N} / b \leq i \leq e \bullet (\sigma_i \oplus \{v \mapsto i\})(s) \Rightarrow \sigma_{i+1}}{\sigma(\text{for } v := \text{beg to end do } s \text{ od}) \Rightarrow \sigma_{\max(b, e+1)}}$$

- This resets the loop variable at the beginning of each iteration
- A static test can prevent assignments to the loop variable

Exercise 8.2(a, b)

$$\frac{\frac{\text{True}}{x \geq -5 \Rightarrow x \geq -6} \text{ (arith.)} \quad \frac{\text{True}}{\{5-x \leq 11\} z := 5-x \{z \leq 11\}} \text{ (assign.)}}{x \geq -5 \Rightarrow 5-x \leq 11 \quad \{5-x \leq 11\} z := 5-x \{z \leq 11\}} \text{ (conseq.)}$$

$$\frac{}{\{x \geq -5\} z := 5-x \{z \leq 11\}}$$

$$\begin{aligned} & \{x \geq -5\} z := 5-x \{z \leq 11 \wedge x \geq -7\} \\ \Leftarrow & \langle \text{(left consequence)} \rangle \\ & (x \geq -5 \Rightarrow 5-x \leq 11 \wedge x \geq -7) \\ & \wedge \{5-x \leq 11 \wedge x \geq -7\} z := 5-x \{z \leq 11 \wedge x \geq -7\} \\ \Leftarrow & \langle \text{(arithmetic, assignment)} \rangle \\ & (x \geq -5 \Rightarrow x \geq -6 \wedge x \geq -7) \wedge \text{True} \\ \Leftarrow & \langle \text{(arithmetic)} \rangle \\ & \text{True} \end{aligned}$$

Exercise 8.2(c)

$$\{x \geq -5\} z := 5 - x \{z \leq 11 \wedge x \geq -3\}$$

Using operational semantics, we can prove a **counterexample**:

$$\frac{\frac{\frac{\{x \mapsto -5\}(5) \Rightarrow 5 \quad \{x \mapsto -5\}(x) \Rightarrow -5}{\{x \mapsto -5\}(5-x) \Rightarrow 10}}{\{x \mapsto -5\}(z := 5-x) \Rightarrow \{x \mapsto -5, z \mapsto 10\}}}$$

This last state clearly does not satisfy $\{z \leq 11 \wedge x \geq -3\}$

Exercise 8.2(d)

$$\begin{aligned} & \{x \geq -5\} z := 5 - x ; x := x + 2 \{z \leq 11 \wedge x \geq -3\} \\ \Leftarrow & \langle \text{sequence rule} \rangle \\ & \{x \geq -5\} z := 5 - x \{z \leq 11 \wedge x + 2 \geq -3\} \\ & \wedge \{z \leq 11 \wedge x + 2 \geq -3\} x := x + 2 \{z \leq 11 \wedge x \geq -3\} \\ \Leftarrow & \langle \text{left consequence, assignment} \rangle \\ & (x \geq -5 \Rightarrow (5 - x \leq 11 \wedge x + 2 \geq -3)) \\ & \wedge \{5 - x \leq 11 \wedge x + 2 \geq -3\} z := 5 - x \{z \leq 11 \wedge x + 2 \geq -3\} \\ & \wedge \text{True} \\ \Leftarrow & \langle \text{logic and arithmetic, assignment} \rangle \\ & (x \geq -5 \Rightarrow x \geq -6) \wedge (x \geq -5 \Rightarrow x \geq -5) \wedge \text{True} \\ \Leftarrow & \langle \text{arithmetic} \rangle \\ & \text{True} \end{aligned}$$

Exercise 8.2(e)

$$\{x \geq -5\} z := 5 - x ; x := x + z \{z \leq 11 \wedge x = 2\}$$

We again use operational semantics (expression evaluation not shown) to prove a **counterexample**:

$$\begin{aligned} \sigma_1 &= \{x \mapsto 0\} \\ \sigma_2 &= \{x \mapsto 0, z \mapsto 5\} \\ \sigma_3 &= \{x \mapsto 5, z \mapsto 5\} \end{aligned}$$

$$\frac{\frac{\sigma_1(5-x) \Rightarrow 5 \quad \sigma_2(x+z) \Rightarrow 5}{\sigma_1(z := 5-x) \Rightarrow \sigma_2} \quad \sigma_2(x := x+z) \Rightarrow \sigma_3}{\sigma_1(z := 5-x ; x := x+z) \Rightarrow \sigma_3}$$

Although $\sigma_1 = \{x \mapsto 0\}$ satisfies the precondition $\{x \geq -5\}$, the final state $\sigma_3 = \{x \mapsto 5, z \mapsto 5\}$ does not satisfy the postcondition $\{z \leq 11 \wedge x = 2\}$.

Exercise 8.2(f)

$$\text{Rule for one-sided conditional: } \frac{\{P \wedge b\} S_1 \{Q\} \quad P \wedge \neg b \Rightarrow Q}{\{P\} \text{ if } b \text{ then } S_1 \text{ fi } \{Q\}}$$

$$\begin{aligned} & \{z = \text{abs}(x)\} \text{ if } x \geq 0 \text{ then } z := -z \text{ fi } \{xz = -x^2\} \\ \Leftarrow & \langle \text{one-sided conditional} \rangle \\ & \{z = \text{abs}(x) \wedge x \geq 0\} z := -z \{xz = -x^2\} \\ & \wedge (z = \text{abs}(x) \wedge x < 0 \Rightarrow xz = -x^2) \\ \Leftarrow & \langle \text{left consequence, arithmetic} \rangle \\ & (z = \text{abs}(x) \wedge x \geq 0 \Rightarrow x \cdot (-z) = -x^2) \\ & \wedge \{x \cdot (-z) = -x^2\} z := -z \{xz = -x^2\} \\ & \wedge (z = -x \Rightarrow xz = -x^2) \\ \Leftarrow & \langle \text{arithmetic, assignment, arithmetic} \rangle \\ & (z = x \Rightarrow x \cdot (-z) = -x^2) \wedge \text{True} \wedge \text{True} \\ \Leftarrow & \langle \text{arithmetic} \rangle \\ & \text{True} \end{aligned}$$

Exercise 8.2(g)

$\{z = 0\}$ **if** $x = 0$ **then** $w := \text{True}$ **else** $z := 1/x$ **fi** $\{\neg w \rightarrow xz = 1\}$
 \Leftarrow \langle conditional \rangle
 $\{z = 0 \wedge x = 0\} w := \text{True} \{\neg w \rightarrow xz = 1\}$
 $\wedge \{z = 0 \wedge x \neq 0\} z := 1/x \{\neg w \rightarrow xz = 1\}$
 \Leftarrow \langle left consequence , left consequence \rangle
 $(z = 0 \wedge x = 0 \Rightarrow (\neg \text{True} \rightarrow xz = 1))$
 $\wedge \{\neg \text{True} \rightarrow xz = 1\} w := \text{True} \{\neg w \rightarrow xz = 1\}$
 $\wedge (z = 0 \wedge x \neq 0 \Rightarrow (\neg w \rightarrow x \cdot (1/x) = 1))$
 $\wedge \{\neg w \rightarrow x \cdot (1/x) = 1\} z := 1/x \{\neg w \rightarrow xz = 1\}$
 \Leftarrow \langle logic , assignment , logic , assignment \rangle
 $(z = 0 \wedge x = 0 \Rightarrow (\text{False} \rightarrow xz = 1)) \wedge \text{True} \wedge (x \neq 0 \Rightarrow x \cdot (1/x) = 1)$
 \Leftarrow \langle *ex falso quodlibet* , arithmetic \rangle
 $(z = 0 \wedge x = 0 \Rightarrow \text{True}) \wedge \text{True}$
 \Leftarrow \langle logic \rangle
 True

Exercise 9.1

$\{\text{True}\}$
 $(i, j, s) := (0, 0, 0);$
while $i \neq n$ **do**

if $i = j$

then $(i, j, s) := (i + 1, 0, s + 1)$

else $(j, s) := (j + 1, s + 2)$

fi

od
 $\{s = n^2 + 2j\}$

Exercise 9.1 — Proof

$\{\text{True}\} (i, j, s) := (0, 0, 0); \text{ while } i \neq n \text{ do } B \text{ od } \{s = n^2 + 2j\}$
 \Leftarrow \langle right consequence \rangle
 $\{\text{True}\} (i, j, s) := (0, 0, 0); \text{ while } i \neq n \text{ do } B \text{ od } \{s = i^2 + 2j \wedge i = n\}$
 $\wedge (s = i^2 + 2j \wedge i = n \Rightarrow s = n^2 + 2j)$
 \Leftarrow \langle sequence , logic \rangle
 $\{\text{True}\} (i, j, s) := (0, 0, 0) \{s = i^2 + 2j\}$
 $\wedge \{s = i^2 + 2j\} \text{ while } i \neq n \text{ do } B \text{ od } \{s = i^2 + 2j \wedge i = n\}$
 $\wedge \text{True}$
 \Leftarrow \langle left consequence , while -rule \rangle
 $(\text{True} \Rightarrow 0 = 0^2 + 2 \cdot 0)$
 $\wedge \{0 = 0^2 + 2 \cdot 0\} (i, j, s) := (0, 0, 0) \{s = i^2 + 2j\}$
 $\wedge \{s = i^2 + 2j \wedge i \neq n\} \text{ if } i = j \text{ then } (i, j, s) := (i + 1, 0, s + 1)$
 $\text{ else } (j, s) := (j + 1, s + 2) \text{ fi } \{s = i^2 + 2j\}$

Exercise 9.1 — Proof (ctd.)

\Leftarrow \langle arithmetic , assignment , conditional \rangle
 $\text{True} \wedge \text{True}$
 $\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i = j\} (i, j, s) := (i + 1, 0, s + 1) \{s = i^2 + 2j\}$
 $\wedge \{s = i^2 + 2j \wedge i \neq n \wedge i \neq j\} (j, s) := (j + 1, s + 2) \{s = i^2 + 2j\}$
 \Leftarrow \langle left consequence , left consequence \rangle
 $(s = i^2 + 2j \wedge i \neq n \wedge i = j \Rightarrow s + 1 = (i + 1)^2 + 2 \cdot 0)$
 $\wedge \{s + 1 = (i + 1)^2 + 2 \cdot 0\} (i, j, s) := (i + 1, 0, s + 1) \{s = i^2 + 2j\}$
 $\wedge (s = i^2 + 2j \wedge i \neq n \wedge i \neq j \Rightarrow s + 2 = i^2 + 2 \cdot (j + 1))$
 $\wedge \{s + 2 = i^2 + 2 \cdot (j + 1)\} (j, s) := (j + 1, s + 2) \{s = i^2 + 2j\}$
 \Leftarrow \langle arithmetic , assignment , arithmetic , assignment \rangle
 $\text{True} \wedge \text{True} \wedge \text{True} \wedge \text{True}$

Exercise 9.1 — Stronger Postcondition

```

{True}
(i, j, s) := (0, 0, 0);
while i ≠ n do
  if i = j
  then (i, j, s) := (i + 1, 0, s + 1)
  else (j, s) := (j + 1, s + 2)
  fi
od
{s = n2}

```

Challenge: How can you prove this?
Can you reformulate the program to make this easier?

Integer Square Root

```

{n ≥ 0}
(a, b) := (0, n + 1);
while a + 1 ≠ b
do  d := (a + b)/2;
   if d * d ≤ n
   then a := d
   else b := d
   fi
od
{a2 ≤ n < (a + 1)2}

```

All variables and expressions are of type **integer**.