Do you remember?

- The while programming language has an operational semantics
- We can code functions over complex domains into $\mathbb{N} \rightarrow \mathbb{N}$
- A function is computable if we can write a while program for it

Something to think about:

- If we run a program and always get a state $\sigma$ such that $\sigma(x) = \sigma(y)^2$, does that program compute $f(x) = y^2$?
- If $x > y$ is true of a state after executing $x := x + 1$ what can we say for certain about the state beforehand?
- What is the logical relationship between $x > 0$ and $x \geq 0$?
I motivated the need for correctness before. Key points:

- Simple bugs in programs can cause loss of reputation, money, and life
- Testing is popular because it is ‘easy’ but it can never be certain
  \textit{Testing shows the presence, not the absence of bugs (Dijkstra)}
- Formal techniques are becoming more popular, especially in safety and security critical domains

There are different kinds of correctness (from simple syntactic checks to reachability in concurrent systems).

We focus on the semantic correctness of showing that a program implements a particular function.
Correctness

To establish whether a program is correct we need:

- A specification of what we want it to do
- A method for checking whether the program meets the specification

Abstractly, a program represents a set of pairs \( \langle s_{\text{start}}, s_{\text{end}} \rangle \) where executing the program on \( s_{\text{start}} \) gives \( s_{\text{end}} \). What does this description miss?

The two big ideas:

1. We can specify the desired behaviour of a program using predicates on these states.
2. We can check this specification by breaking down how each expression changes the states and hence the predicates
Predicates on States (useful later)

Remember, a state is a total function from variables to numbers.

A predicate on a state is a function from states to booleans.

We can view a predicate as denoting a possibly infinite set of states e.g. $x > 0$ denotes $[x \mapsto 1], [x \mapsto 2], [x \mapsto 3], \ldots$

We will abuse notation and sometimes treat a predicate as the state it denotes e.g. $[x \mapsto 1, y \mapsto 0] \in x > y$ and $[x \mapsto 0] \notin x > 0$.

Logical relationships on predicates lift to the sets of states they denote e.g. $x > 0 \rightarrow x \geq 0$ and hence $\{\sigma \mid (x > 0)(\sigma)\} \supseteq \{\sigma \mid (x \geq 0)(\sigma)\}$. 
Consider the problem

*Take as input a first number* $c$ *which is one of* $-1$, $0$, or $1$ *and a second number* $x$ *(which is an integer). If* $c = 0$ *return* $x$, *if* $c = -1$ *return* $-|x|$, *and if* $c = 1$ *return* $|x|$.

How should we specify it mathematically such that any and all programs satisfying the specification solve the problem?
Consider the problem

Take as input a first number $c$ which is one of $-1$, $0$, or $1$ and a second number $x$ (which is an integer). If $c = 0$ return $x$, if $c = -1$ return $-|x|$, and if $c = 1$ return $|x|$.

How should we specify it mathematically such that any and all programs satisfying the specification solve the problem?

We will provide two predicates on the input and the output e.g.

1

2

does this work?

The idea is that for any input state satisfying 1 a correct program should produce an output state satisfying 2.
Consider the problem

*Take as input a first number* $c$ *which is one of* $-1$, 0, or 1 *and a second number* $x$ *(which is an integer). If* $c = 0$ *return* $x$, *if* $c = -1$ *return* $-|x|$, *and if* $c = 1$ *return* $|x|$.  

How should we specify it mathematically such that any and all programs satisfying the specification solve the problem?

We will provide two predicates on the input and the output e.g.

1. $(c = -1) \lor (c = 0) \lor (c = 1)$
2. $(c = -1 \rightarrow r = -|x|) \land (c = 0 \rightarrow r = x) \land (c = 1 \rightarrow r = |x|)$

does this work?

The idea is that for any input state satisfying 1 a correct program should produce an output state satisfying 2.
Consider the problem

*Take as input a first number $c$ which is one of $-1$, $0$, or $1$ and a second number $x$ (which is an integer). If $c = 0$ return $x$, if $c = -1$ return $-|x|$, and if $c = 1$ return $|x|$.*

How should we specify it mathematically such that any and all programs satisfying the specification solve the problem?

We will provide two predicates on the input and the output e.g.

1. $(c = -1) \lor (c = 0) \lor (c = 1)$
2. $(c = -1 \rightarrow r = -|x|) \land (c = 0 \rightarrow r = x) \land (c = 1 \rightarrow r = |x|)$

does this work? (we will return to a special case where it doesn’t later)

The idea is that for any input state satisfying 1 a correct program should produce an output state satisfying 2.
We introducing the following notion of a specification of program $S$

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

where $P$ is a precondition and $Q$ is a postcondition

The meaning of this is that if $P(\sigma_{start})$ holds then $Q(\sigma_{end})$ should hold

We can define the set of states a predicate denotes i.e. $P_\sigma \equiv \{ \sigma \mid P(\sigma) \}$

If $\{ P \} \ S \ \{ Q \}$ is true then $S$ captures a function from $P_\sigma$ to $Q_\sigma$

The above notation is a Hoare Triple after Tony Hoare who invented it
Property: find the maximum of $x$ and $y$ and place it in $z$

Specification: ?
Property: find the maximum of $x$ and $y$ and place it in $z$

Specification: \{ \} S \{ z = \max(x, y) \} 

Let

$$\max(x, y) = \begin{cases} x & \text{if } x > y \\ y & \text{otherwise} \end{cases}$$

or equivalently

$$(z = \max(x, y)) \iff (z \geq x \land z \geq y \land (z = x \lor z = y))$$

or also equivalently

$$(z = \max(x, y)) \iff ((x \geq y) \rightarrow z = x) \land ((y > x) \rightarrow z = y)$$
Property: find the maximum of $x$ and $y$ and place it in $z$

Specification: \{ \} S \{ z = \max(x, y) \} 

Let $S$ be

$$x := 0; \quad y := 0; \quad z := 0$$

We now have $0 = \max(0, 0)$ but does the program satisfy the property?

What went wrong?
Property: find the maximum of \( x \) and \( y \) and place it in \( z \)

Specification: \( \{ x = a \land y = b \} \quad S \quad \{ z = \max(a, b) \} \) where \( a, b \) not in \( S \)

Let \( S \) be

\[
\begin{align*}
x & := 0; \quad y := 0; \quad z := 0
\end{align*}
\]

We now have \( 0 = \max(a, b) \) which does not hold for all \( a \) and \( b \).
**Property**: find the maximum of $x$ and $y$ and place it in $z$

**Specification**: $\{ x = a \land y = b \} \implies S \{ z = \max(a, b) \} \text{ where } a, b \text{ not in } S$

Let $S$ be

`if x>y then z:=x else z:=y`

We should usually introduce auxiliary variables or argue that we do not need them, how?
Property: Given values in variables $x, y \in \mathbb{N}$ such that $y \neq 0$ set $d$ and $r$ such that $x = d \times y + r$ and $r < y$

Specification: ?
Writing Some Specifications

Property: Given values in variables $x, y \in \mathbb{N}$ such that $y \neq 0$ set $d$ and $r$ such that $x = d \times y + r$ and $r < y$

Specification:

$$\{ x \geq 0 \land y > 0 \} \quad S \quad \{ x = d \times y + r \land 0 \leq r < y \}$$

Why $y > 0$? Why $0 \leq r$?
An Informal Argument

Let us consider the program

```plaintext
if x > y then z := x else z := y
```

Is it correct with respect to \( \{ \} S \{ z = \max(x, y) \} \)?

Given any state \( s_{start} \) what can we say about \( s_{end} \)? There are two possible paths through the program:

- If \( x > y \) then we perform \( z := x \), after which \( z = x ∧ z > y \)
- If \( \neg(x > y) \) then we perform \( z := y \), after which \( z = y ∧ z ≥ x \)

So we know that this holds after the program

\[
(x > y \rightarrow z = x ∧ z > y) ∧ (y ≥ x \rightarrow z = y ∧ z ≥ x)
\]

which is a reasonable definition of \( z = \max(x, y) \)

Next we do this formally. We look at loop-free programs first so that we can become familiar with the ideas in the setting of simple programs.
A Formal Approach

Shortly I will introduce a set of axiomatic rules for proving correctness. But we need to separate two kinds of correctness

- **Partial Correctness.** A program is partially correct if whenever it terminates it satisfies the specification
- **Total Correctness.** A program is totally correct if it satisfies the specification and always terminates

We will use

\[
\{ P \} S \{ Q \}
\]

to mean partial correctness and

\[
[ P ] S [ Q ]
\]

to mean total correctness.
The following statement is true

\[ \{ \} \text{ while } \text{true} \text{ do skip } \{ \text{false} \} \]

as the program never terminates. This is also true

\[ \{ \} \text{ if } x > y \text{ then } z := x \text{ else } (\text{while } \text{true} \text{ do skip}) \{ z = \max(x, y) \} \]

as whenever it terminates it satisfies the specification! This is also true

\[ [x \neq x] S [\ ] \]

for any program \( S \) as \( x \neq x \) will not hold any states.

This shows some of the limitations of this approach
Axiomatic System

\[ \text{ass}_p \quad \{ P[x \mapsto A[a]] \} \ x := a \ { P } \]

\[ \text{skip}_p \quad \{ P \} \ \text{skip} \ \{ P \} \]

\[ \text{comp}_p \quad \{ P \} \ S_1 \ { Q } , \ \{ Q \} \ S_2 \ { R } \]
  \[ \{ P \} \ S_1; \ S_2 \ { R } \]

\[ \text{if}_p \quad \{ P \land B[b] \} \ S_1 \ { Q } , \ \{ P \land \neg B[b] \} \ S_2 \ { Q } \]
  \[ \{ P \} \ \text{if} \ b \ \text{then} \ S_1 \ \text{else} \ S_2 \ { Q } \]

\[ \text{while}_p \quad \{ P \land B[b] \} \ S \ { P } \]
  \[ \{ P \} \ \text{while} \ b \ \text{do} \ S \ { P \land \neg B[b] } \]
The obvious ones (see them in action soon)

Skipping doesn’t change anything

\[
\{ P \} \text{skip} \{ P \}
\]

For sequence \( S_1; S_2 \), the postcondition of \( S_1 \) needs to make the precondition of \( S_2 \) true

\[
\begin{align*}
\{ P \} & \quad S_1 \{ Q \}, \quad \{ Q \} \quad S_2 \{ R \} \\
\{ P \} & \quad S_1; \quad S_2 \{ R \}
\end{align*}
\]

For conditionals we can assume the condition and check the statement

\[
\begin{align*}
\{ P \land B[b] \} & \quad S_1 \{ Q \}, \quad \{ P \land \neg B[b] \} \quad S_2 \{ Q \} \\
\{ P \} & \quad \text{if } b \text{ then } S_1 \text{ else } S_2 \{ Q \}
\end{align*}
\]
People often find this rule counterintuitive

\[ \{ P[x \mapsto A[a]] \} \ x := a \ \{ P \} \]

as it feels backwards. The handout discusses why alternatives don’t work.

Let’s return to the earlier question

If \( x > y \) is true of a state after executing \( x := x + 1 \) what can we say for certain about the state beforehand?
People often find this rule counterintuitive

\[
\{ P[x \mapsto A[a]] \} \quad x := a \quad \{ P \}
\]

as it feels backwards. The handout discusses why alternatives don’t work.

Let’s return to the earlier question

If \( x > y \) is true of a state after executing \( x := x + 1 \) what can we say for certain about the state beforehand? \( x + 1 > y \) is true
People often find this rule counterintuitive

\[ \{ P[x \mapsto A[a]] \} \ x := a \ \{ P \} \]

as it feels backwards. The handout discusses why alternatives don’t work.

Let’s return to the earlier question

If \( x > y \) is true of a state after executing \( x := x + 1 \) what can we say for certain about the state beforehand? \( x + 1 > y \) is true

Consider two states \( \sigma_1 \) and \( \sigma_2 \) such that

\[ \langle \sigma_1, x := a \rangle \Rightarrow \sigma_2 \]

We want to find a predicate that is true for \( \sigma_1 \). The states \( \sigma_1 \) and \( \sigma_2 \) only differ in the value of \( x \). If \( P \) is true for \( \sigma_2 \) then we know it is true when \( x \) is replaced by the value for \( a \).
Hints on Applying Assignment Rule

Apply it \textit{backwards} i.e. start with the postcondition and apply the assignment to get a precondition. If you already have a precondition then fix that afterwards (see later).

This suggests that we start at the end of the proof and move back to the start. This is roughly true - see tomorrow for my \textit{general formula} for creating partial correctness proofs.

\textbf{Examples} next.
Some Examples

\{ x \geq 0 \} \ x := x + 1 \ \{ x > 0 \} \\

\{ \} \ x := 0; y := 0 \ \{ x = y \} \\

\{ x \neq y \} \text{if } y > x \text{ then } t := y; y := x; x := t \text{ else skip} \{ x > y \land x \neq y \} \\

\{ x = a \land y = b \} \ z := x; z := z + y \ \{ z = a + b \}