CSE 331: Floyd-Hoare Logic

An **assertion** is a true/false claim about the values of the variables of a program. Examples: `x` is positive, array `a` is sorted.

An assertion Q **holds** at a program point L, if Q is true on all executions of the program that reach L.

We use the notation `{{ Q }}` to mean "assertion Q holds here".
This avoids the need to name the program point L_1 etc.
Example at right: $n \geq 0$ holds after the if statement.

Many different assertions can hold at the same point.
Which should we write down?
Example at right: $n \geq -1$ also holds at that point.

```
// @return the absolute value of n
private int abs(int n) {
    if (n < 0) {
        n = -n;
    }
    {{ n ≥ 0 }}    // L1
    return n;
}
```

Generally, we are interested in the **strongest** assertion that holds at a program point.
In the example above, since $n \geq 0$ is stronger than $n \geq -1$, so we only write $n \geq 0$.
Is $n \geq 0$ really the *strongest possible* assertion that holds at that point? _____

A **Hoare triple** `{{ P }} c {{ Q }}` is the combination of a precondition P, some code snippet c, and a postcondition Q.
Allows giving pre/post-conditions to code other than an entire method.

A Hoare triple is **valid** if: for any starting state satisfying P, after running c from that state, Q will hold.
This is essentially the definition of method correctness but for any snippet of code, not just an entire method.

Examples of validity/invalidity:

```
{{ true }}
x = 5;
{{ x = 5 }}
```

```
{{ x = 3 }}
y = x + 1;
{{ y = 4 }}
```

```
{{ x = 3 }}
y = x + 1;
{{ y > 0 }}
```

```
{{ x = 10 }}
x = x - 4;
{{ x > 0 }}
```

```
{{ true }}
y = x + 1;
{{ y = 4 }}
```

```
{{ x < 0 }}
y = x - 1;
{{ x < 0 }}
```

```
{{ x > 0 }}
x = x - 1;
{{ x > 0 }}
```

```
{{ false }}
x = 0;
{{ x = 1 }}
```

Floyd Logic is a set of techniques for reasoning about Hoare triples and their validity.

Forward reasoning: Given P and c, find the strongest postcondition so that `{{ P }} c {{ Q }}` is valid.

Forward reasoning for assignment statements:

```
{{ x = 3 }}
y = x + 1;
{{ ____________________ }}
```

Forward reasoning when changing a variable:

```
{{ x = 3 }}
x = x + 1;
{{ ____________________ }}
```

Forward reasoning for conditionals:

```
{{ true }}
if (n < 0) {
    {{ ____________________ }}
    n = -n;
    {{ ____________________ }}
} else {
    {{ ____________________ }}
    // do nothing
    {{ ____________________ }}
}
{{ ____________________ }}
```

Backward reasoning: Given Q and c, find the *weakest* precondition so that `{{ P }} c {{ Q }}` is valid.

Backward reasoning for assignment statements:

```
{{ ____________________ }}
x = x + 1;
{{ x = 4 }}
```