# CSE 331 Software Design and Implementation

## Section 1: *Reasoning About Code, Part I*

Contents:



## Introduction

The primary purpose of this handout is to make available the solutions to section exercises. It assumes you are already familiar with concepts presented in the lecture slides and assigned readings. **Not all of the exercises in this handout were covered in section.** This quarter, the section focused on backward reasoning through if/else statements.

## Forward Reasoning

Given a program consisting of a precondition, a sequence of code statements, and a postcondition, we want to determine if the program is correct. A "correct" program is one where the postcondition is guaranteed to be true after the code executes if the precondition was true beforehand.

One way to determine this is with **forward reasoning**. In forward reasoning, we start with the precondition and work our way to the postcondition, writing down what facts must be true after each statement.

### Exercise

Use forward reasoning to determine whether the following program is correct. Assume all variables are integers.

```
{x >= 0}
x = x+1;
y = 2x;
z = y-x;
{z > 0}
```

### Solution

Annotate the code with an assertion after each statement:

```
{x >= 0}
x = x+1;
{x >= 1}
y = 2x;
{x >= 1 && y == 2x}
z = y-x;
{x >= 1 && y == 2x && z = y-x}
  => {x > 0 && z == x}
  => {z > 0}
```

Yes, this code is correct. Forward reasoning shows that the postcondition `z > 0` is guaranteed to be true.

Even in such a short block of code, our assertions get long by the end. We don't know what information we'll need later on, so we have to keep track of it all. Any more statements and variables and we'll end up with cumbersome assertions that are inconvenient to write and allow the information we need to get hidden among all the cruft.

## Backward Reasoning

**Backward reasoning**, among other things, keeps assertions small and simple. With backward reasoning, we start at the end of the block of code and work our way up to find a precondition that guarantees the postcondition.

### Exercise

Show that the previous program is correct, this time using backward reasoning.

### Solution

Annotate the code with an assertion before each statement:

```
{x+1 > 0} => {x >= 0} (assume ints, as stated earlier)
x = x+1;
{2x-x > 0} => {x > 0}
y = 2x;
{y-x > 0}
z = y-x;
{z > 0}
```

Again, we have shown that the precondition `x >= 0` will guarantee the postcondition `z > 0`.

## Weakest Precondition

Backward reasoning allows us to find the **weakest precondition**: the most general precondition (i.e. the broadest set of inputs) that guarantees the postcondition.

### Exercises

Using the rules given in the reading and lecture, find the weakest precondition for each of the following programs. Assume all variables are integers.

```
x = y;          y = y+3;      z = x-y+2;       y = Math.sqrt(w);    y = x;
x = x+1;        x = 2y;       z = 3z-6;        x = 2y;              y = y+1;
{x >= 0}        z = x+8;      {z != 0}         x = x+1;             {y > x}
                {z > 2w}                       {-5 < x < 5}
```

For these next exercises, recall that `wp("if (b) S1 else S2", Q) = (b ^ wp("S1", Q) V -b ^ wp("S2", Q))`.

```
if (x < 0) {
    y = -x;
else {
    y = x;
}
{ y = |x| }
```

```
y = x + 4;
if (x > 0) {
    y = x*x - 1;
} else {
    y = y + x;
}

// This line could cause a divide-by-zero error. Let's find out how.
x = x / y;
```

```
if (x == 0) {
    res = 0;
} else {
    if (x > 0) {
        x = -x;
    } else {
        // Do nothing
    }
    res = x*x*x;
}
{ res > -1 }
```

## Solutions

```
    {y >= -1}       {y+7 > w}       {x != y}
    x = y;          y = y+3;        z = x-y+2;
    {x >= -1}       {2y+8 > 2w}     {z != 2}
    x = x+1;        x = 2y;         z = 3z-6;
    {x >= 0}        {x+8 > 2w}      {z != 0}
                    z = x+8;
                    {z > 2w}



    {0 <= w < 4}            {x+1 > x} -> {true}
    y = Math.sqrt(w);       y = x;
    {-3 < y < 2}            {y+1 > x}
    x = 2y;                 y = y+1;
    {-6 < x < 4}            {y > x}
    x = x+1;
    {-5 < x < 5}
```

```
{ (x<0 & -x = |x|) | (x>0 & x = |x|) } <-> {true}
if (x < 0) {
    { -x = |x| }
    y = -x;
    { y = |x| }
else {
    { x = |x| }
    y = x;
    { y = |x| }
}
{ y = |x| }
```

```
{ x=1 | (x<=0 & x+4=-x) }  <->  { x=1 | x=-2 }
y = x + 4;
{ (x>0 & (x=1 | x=-1)) | (x<=0 & y=-x) }  <->  { x=1 | (x<=0 & y=-x) }
if (x > 0) {
    {x*x - 1 = 0}  <->  {x = 1 | x = -1}
    y = x*x - 1;
    { y = 0 }
} else {
    { y+x = 0 }  <->  {y = -x}
    y = y + x;
    { y = 0 }
}
{ y = 0 }
// This line could cause a divide-by-zero error. Let's find out how.
x = x / y;
```

```
{ (-|x|)^3>-1 & x!=0 | x==0 & true }   <->   { (-|x|)^3>-1 }
if (x == 0) {
    { true }
    res = 0;
    { res > -1 }
} else {
    {x>0 & (-x)^3>-1  |  x<=0 & x^3>-1}   <->   { (-|x|)^3>-1 }
```

```
    if (x > 0) {
        { (-x)^3 > -1 }
        x = -x;
        { x^3 > -1 }
    } else {
        { x^3 > -1 }
        // Do nothing
    }
    { x^3 > -1 }
    res = x*x*x;
    { res > -1 }
}
{ res > -1 }
```