CSE 331 Spring 2012 Homework Assignment 1

Getting Started

You will need to follow a couple of steps to set up for the homework sequence this quarter. **Complete these steps by the end of the week** so the staff can help you work through any problems.

1. Run the student-setup script as explained at http://www.cs.washington.edu/education/courses/cse331/12sp/tools/overview.html#initial-setup/.
2. Check out your Subversion repository**.** This quarter, you will receive starter code and submit your assignments through Subversion (SVN).
   - You will learn more about SVN in section. Also see the version control handout at http://www.cs.washington.edu/education/courses/cse331/12sp/tools/versioncontrol.html.
   - Follow the directions in the version control handout (above) to check out your repository. If you are working on your own computer, also see the SVN section of the Working At Home handout at http://www.cs.washington.edu/education/courses/cse331/12sp/tools/WorkingAtHome.html#Step_4
   - In the src/hw1/ directory of your repository, you will find a copy of this assignment in PDF form.

Turnin

- Homework 1 is due Tuesday, April 3, 2012 at 11:00 PM.
- Please submit your answers by adding and committing a single PDF file named hw1_answers.pdf to the src/hw1/answers/ directory of your repository. Follow the directions in the version control handout for adding and committing files.
- Your file should be no larger than 3MB. Scanned copies of hand-written documents are fine, as long as they are legible when printed.
- **IMPORTANT!!!** After completing your homework, it is STRONGLY recommended that you run the validator tool as described at http://www.cs.washington.edu/education/courses/cse331/12sp/tools/turnin.html#validate. This tool checks for common structural mistakes, such as naming the file incorrectly or neglecting to add and commit it to your repository correctly (an all-too-common mistake for students new to version control).
- **Submissions that do not follow these guidelines will not be graded.**

Other Information for HW1

- In your answers, you may use any standard symbols for “and” and “or” (& and |, V and Λ, etc.).
- If no precondition is required for a code sequence, simply write {true} to denote the trivial precondition.
- You may assume that integer overflow will never occur.

1. **Forward reasoning with assignment statements.** Find the strongest postcondition of each code sequence by inserting the appropriate assertion in each blank. The first assertion in part (a) is supplied as an example.

   a.
   ```
   { x > 0 }
   x = 10;
   { x == 10 }
   y = 2 * x;
   {x = 10 & y = 20 & y = 2*x }
   z = y + 4;
   {x = 10  &  y = 20  &  z = 24  &  y = 2*x  &  z = y+4}
   x = z / 2;
   {x = 12  &  y = 20  &  z = 24  &  z = y+4  &  x = z/2}
   y = 0;
   {x = 12  &  y = 0  &  z = 24  &  x = z/2}
   ```

   b.
   ```
   { x > 0 }
   y = x;
   {x > 0  &  y > 0  &  y = x}
   y = y + 2;
   {x > 0  &  y > 2}
   ```

   c.
   ```
   { |x| > 10 }
   x = -x;
   {|x| > 10}
   x = x / 2;
   {|x| > 5}
   x = x + 1;
   {x < -4  |  x > 6}
   ```

   d.
   ```
   { y > 2x }
   y = y * 2;
   {y > 4x}
   x = x + 1;
   {y > 4(x-1)}
   ```

2. **Backward reasoning with assignment statements.** Find the weakest precondition of each code sequence by inserting the appropriate assertion in each blank.

   a.
   ```
   {x > 0}
   ```

```
x = x + 5;
{y > 10  &  x > 5}
y = 2 * x;
{ y > 10 }
```

b.

```
{x <= -3}
y = x + 6;
{x+y <= 0}
z = x + y;
{ z <= 0 }
```

c.

```
{2x > w-10}
y = w – 10;
{2x > y}
x = 2 * x;
{ x > y }
```

d.

```
{2 > s  &  w > 0}
t = 2 * s;
{2 > s  &  2s+w > t}
r = w + 4;
{r > 2s+w  &  2s+w > t}
s = 2*s + w;
{ r > s & s > t }
```

3. **Backward reasoning with if/else statements.** Find the weakest precondition for the following conditional statement using backward reasoning, inserting the appropriate assertion in each blank. Be sure to explicitly verify that the intermediate postconditions for the two cases imply the total postcondition, i.e. show that (Q1 | Q2) => Q.

```
{(x >= 0  &  x != 0)  |  (x < 0  &  x != 1)}
if (x >= 0)
        {x != 0}
        z = x;
        {z != 0}
else
        {x != -1}
        z = x + 1;
        {z != 0}
{z != 0 }
```

4. **Wrong weakest precondition for if/else statements.** Willy Wazoo says that the CSE 331 formula for wp(“If C, S1 else S2”, Q) is too hard to remember. He suggests this simpler definition:

   wp(“If C, S1 else S2”, Q) = wp(S1, Q) ∨wp(S2, Q) .

   Consider the following Hoare triple:

```
P:  {x < 3  |  x < -1}
if (x == 0)
{
    x = x - 5;      // S1
} else {
    x = x*2;     // S2
}
Q:  {x < -2}
```

   a. Fill in S1, S2 with statements of your choice (but read the rest of this problem before making your choice).

   b. Fill in Q with a condition of your choice. Compute P according to Willy’s definition.

   c. In the space below, give a concrete initial value of x that satisfies P, and the corresponding final value of x (resulting from executing the code) that does not satisfy Q. This shows that Willy’s definition is incorrect.

   x = 1

5. **Another wrong weakest precondition for if/else statements.** Willy accepts your counterexample, but he still believes he can simplify the formula. He proposes to try the equation:
   wp("If C, S1 else S2", Q) = wp(S1, Q) $\wedge$wp(S2, Q) .
   Consider the following Hoare triple:

```
P:  {x < 12  &  x > -2}
if (x == 0)
{
    x = x / 2;      // S1
} else {
    x = -3 * x;     // S2
}
Q:  {x < 6}
```

   - **a.** Fill in S1, S2 with statements of your choice (but read the rest of this problem before making your choice).
   - **b.** Fill in Q with a condition of your choice. Compute P according to Willy's definition.
   - **c.** In the space below, give a concrete initial value of x that does not satisfy P, and the corresponding final value of x (resulting from executing the code) that satisfies Q. This shows that Willy's definition does not give the weakest precondition.

   x = 24

6. **Weakest conditions.** Circle the weakest condition in each set.
   a. {x == 20 } { x > 10 } [✓ { x >= 10 }]

   b. { t == 2 } [✓ { t != 0 }] { t > 0 }

   c. { x > 0 & y > 0 } [✓ { x > 0 | y > 0 }]

   d. [✓ { |x+y| > w }] { x+y > w }

7. **Hoare triples.** State whether each Hoare triple is valid. If it is invalid, explain why and show how you would modify the precondition or postcondition to make it valid.

a.

```
{ x < 0 }
y = 2*x;
{ y <= 0 }
```

Valid

b.

```
{ x >= y }
z = x – y;
{ z > 0 }
```

Invalid, because if x = y, then z = 0. This can be modified by changing the precondition to {x > y}

c.

```
{true}
if (x % 2 == 0)
        y = x;
else
        y = x+1;
{ y is even }
```

Valid

d.

```
{ x < 0 }
if (x < 100)
        x = -1;
else
        x = 1;
{ x < 0 }
```

Valid

8. **Verifying correctness.** For each block of code, fill in the intermediate assertions, then use them to state whether the precondition is sufficient to guarantee the postcondition. If it the precondition is insufficient, explain why and indicate where the assertions don't match up.
   (Hint: for assignment statements, use backward reasoning to find the weakest precondition that guarantees the postcondition, then see if the given precondition is weaker than the weakest precondition. For if/else statements, you may find a combination of forward and backward reasoning most useful. Follow the rules given in class for what assertion to insert at each point.)

   a.
   ```
   { x > 0 }
   y = x – 1;
   {y > 0}
   z = 2 * y;
   {z > 0}
   z = z + 1;
   { z > 1 }
   ```
   The precondition is insufficientbecause with backward reasoning, we arrive with the precondition of {x > 1}. This means that a value of 1 for x will not end up wth {z > 1}

   b.
   ```
   { 2x >= w }
   y = w – 2;
   {2*x-2 >= y}
   x = 2*x;
   {x-2 >= y}
   z = x – 2;
   { z >= y }
   ```
   The precondition is sufficient.

   c.
   ```
   { y > 0 }
   if (x == y)
           {y > 0  &  x > 0  &  x = y}
           x= -1;j
           {y > 0  &  x = -1}
   else
           {y != x  &  x > 0}
           x = y - 1;
           {x = y - 1}
   { x < y }
   ```
   The precondition is sufficient.

9. **Write and prove code.** Write a block of code that calculates the smallest even number greater than or equal to x and stores it in y. In other words, y will be assigned either x or x+1. Assume x and y have already been initialized, and annotate your code with assertions before and after each statement to prove that it is correct. At the end of the block, it should be true that y is even and that y == x or y == x +1.

```
{true}
if (x % 2 == 0)
    {x is even}
    y = x;
    {x is even  &  y = x  &  y is even}
else
    {x is odd}
    y = x + 1;
    {x is odd  &  y = x + 1  &  y is even}
{(y = x  &  y is even)  |  (y = x + 1  &  y is even)}
```