15-122 : Principles of Imperative Computation, Spring 2016

Written Homework 1

Due: Tuesday 19th January, 2016

Name: _____

Andrew ID: _____

Section: _____

This written homework is the first of two homeworks that will introduce you to the way we reason about C0 code in 15-122.

Print out this PDF double-sided, staple pages in order,
and write your answers *neatly* by hand in the spaces provided.

The assignment is due by 5:30pm on Tuesday 19th January, 2016.

You can hand in the assignment in during lecture
or in the box outside of Rob Simmons's office (GHC 9101)
between 12:30pm and 5:30pm.

This is an exception because of MLK day:
the usual deadline will be Monday at 5:30.

You must hand in your homework yourself;
do not give it to someone else to hand in.

(This page intentionally left blank. Remember to print double-sided!)

Question	Points	Score
1	2	
2	4	
3	5	
4	4	
Total:	15	

2pts 1. **Running C0 programs**

The file `bar.c0` contains a function `bar` that takes an integer argument and returns an integer. Additionally, there is a `bar-test.c0` file that contains the following:

```
1 int main() {
2   return bar(15122);
3 }
```

(a) From the command line, show how to display the value returned by `bar(15122)` using the C0 compiler.

Solution:

(b) From the command line, show how to display the value returned by `bar(15122)` using the C0 interpreter.

Solution:

4pts

2. Preconditions and postconditions

For the following functions, **either** check the box that says the postcondition always holds when the function is given inputs that satisfy its preconditions **or** give a concrete counterexample: specific values of the inputs such that the preconditions (if there is one) holds and the postcondition does not hold. You don't have to write any proofs.

```
1 int f1(int x, int y)
2 //@requires 0 <= x && x < y;
3 //@ensures \result >= 0;
4 {
5   return y - x;
6 }
```

`@ensures` always true: ☐

x = ☐ y = ☐

```
1 int f2(int x)
2 //@requires x % 2 == 0;
3 //@ensures x < 0 || \result < x;
4 {
5   return x / 2;
6 }
```

`@ensures` always true: ☐

x = ☐

```
1 int f3(int x, int y)
2 //@requires y > 0;
3 //@ensures \result < y;
4 {
5   return x % y;
6 }
```

`@ensures` always true: ☐

x = ☐ y = ☐

```
1 int f4(int x, int y)
2 //@requires x + y == 5;
3 //@ensures \result - x == y;
4 {
5   return 5;
6 }
```

`@ensures` always true: ☐

x = ☐ y = ☐

```
1  int f5(int x, int y)
2  //@ensures \result < 0;
3  {
4    if (x > 0) x = -x;
5    if (y > 0) y = -y;
6    if (y < x) {
7      return y - x;
8    } else {
9      return x - y;
10   }
11 }
```

`@ensures` always true: ☐

x = ☐ y = ☐

```
1  int f6(int x, int y)
2  //@ensures \result >= 0;
3  {
4    if (x >= 0) x = -x;
5    if (y >= 0) y = -y;
6    if (y <= x) {
7      return y - x;
8    } else {
9      return x - y;
10   }
11 }
```

`@ensures` always true: ☐

x = ☐ y = ☐

3. **Thinking about loops**

When we think about loops in 122, we will always concentrate on a single iteration of the loop. A loop will almost always modify something; the following loop modifies the local assignable `i`.

```
1 while (i < n) {
2   i = i + 4;
3 }
```

In order to reason about the loop, we have to think about the two different values stored in the local assignable `i`.

We use the variable i to talk about the value stored in the local `i` before the loop runs (before the loop guard is checked for the first time).

We use the "primed" variable i' to talk about the value stored in the local `i` after the loop runs exactly one time (before the loop guard is next checked).

2pts (a) Consider the following loop:

```
1 while(i < n) {
2    k = j + k;
3    j = j * 2 + i;
4    i = i + 1;
5 }
```

- If $i = 7$, $j = 3$, and $k = 9$, then assuming $7 <$ `n`,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

- If $i = 2y$, $j = x - y$, and $k = y$, then assuming $2y <$ `n`, in terms of x and y,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

- If $j = k$, then assuming $i <$ `n`, in terms of i and k,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

- In general, assuming $i < n$, then in terms of i, j, and k,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

Note that we always say "assuming (something) < `n`," because if that were not the case then the loop wouldn't run, and it wouldn't make any sense to be talking about the values of the primed variables.

1pt (b) Consider this loop:

```
while(...) {
   i = i + 3;
   j = j * 2 + i;
   k = k + i - j;
}
```

Be careful, it looks similar but is trickier! Give simplified answers.

- If $i = 7$, $j = 3$, and $k = 9$, then assuming the loop guard evaluates to true,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

- In general, assuming the loop guard evaluates to true, then in terms of i, j, and k,

  $i' =$ [ ] $j' =$ [ ], and $k' =$ [ ],

2pts (c) Consider this loop:

```
while(a > 0 && b > 0) {
   if (a > b) {
      a = a-b;
   } else {
      b = b-a;
   }
}
```

- If $a = 94$ and $b = 12$, then

  $a' =$ [ ] and $b' =$ [ ]

- If $a = x + y$ and $b = x$, where x and y are both positive integers, then

  $a' =$ [ ] and $b' =$ [ ]

- If $a = x$ and $b = x + z$, where x is a positive integer and z is a non-negative integer, then

  $a' =$ [ ] and $b' =$ [ ]

- If $a > 0$ and $b > 0$, one of the two cases above will always be the case. Therefore, we can conclude which of the following about the values stored in `a` and `b` after an arbitrary iteration of the loop? (Check **all** that apply)

  [ ] $a' \geq 0$ and $b' \geq 0$

  [ ] $a' > 0$ and $b' \geq 0$

  [ ] $a' \geq 0$ and $b' > 0$

  [ ] $a' > 0$ and $b' > 0$

4. **Proving a function correct**

In this question, we'll do part of the proof of correctness for a `compute_sum` relative to a specification function `SUM`. We won't prove that the loop invariants are true initially, and we won't prove that they're preserved by an arbitrary iteration of the loop.

```
1 int compute_sum(int n) {
2   int total = 0;
3   while (n > 0) {
4      total = total + n;
5      n = n - 1;
6   }
7   return total;
8 }
```

1pt (a) Complete the specification function below with the simple mathematical formula that gives the sum of numbers from 0 to n.

```
1 int SUM(int n)
2 //@requires 0 <= n && n < 10000;
3 {
4   return ____________________________________________;
5 }
```

Give a postcondition for `compute_sum` using this specification function.

```
1  int compute_sum(int num_ints)
2  //@requires 0 <= num_ints && num_ints <= 10000;
3
4  //@ensures ____________________________________________;
5  {
6    int n = num_ints;
7    int total = 0;
8    while (n > 0)
9    //@loop_invariant 0 <= n;
10   //@loop_invariant n <= 10000;
11   // Additional loop invariant will go here
12   {
13      total = total + n;
14      n = n - 1;
15   }
16   return total;
17 }
```

Note: in the real world we wouldn't have an efficient closed-form solution used as a specification function for an inefficient loop-based solution. We usually use the slow, simple version as the specification function for the fast one!

2pts (b) Why was it necessary to add the new local `num_ints` in the second version of `compute_sum` above?

Give a suitable extra invariant that would allow us to prove the function correct.

```
//@loop_invariant ____________________________________;
```

Which line numbers would we point to to justify that `n == 0` when the loop terminates?

Substitute in 0 for `n` in your loop invariant on line `15` and then simplify.

When you substitute `\result` for `total` in the simplified version, you should have exactly the postcondition on line `8`. This proves that the loop invariant and the negation of the loop guard imply the postcondition.

1pt (c) Termination arguments for loops (in this class, at least) must have the following form:

During an arbitrary iteration of the loop, the quantity ____ gets strictly larger, but from the loop invariants, we know this quantity can't ever get bigger than ____.

or

During an arbitrary iteration of the loop, the quantity ____ gets strictly smaller, but from the loop invariants, we know this quantity can't ever get smaller than ____.

Assuming that your loop invariants are true initially preserved by every iteration of the loop (which we didn't prove), why does the loop in `compute_sum` terminate?

During an arbitrary iteration of the loop, the quantity

but from the loop invariants, we know that this quantity can't ever get