CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 2b

1. Using "experimental semantics" means running an implementation. Running version 2.12.8 of the `scala` interpreter on the example

```
for (i <- 10 to 9 by 1) print(i)
```

produces no output, indicating that the loop body is not executed, which is the result we probably expected. We can probably be fairly confident that this answer generalizes—after all, there is nothing special about `10` and `9`—but we might want to try some other examples just to make sure. (And we should certainly try an example like `1 to 10` to make sure that something *does* get printed in this case!)

But, remembering what happened in the FORTRAN history, we might be concerned that this behavior is specific to this particular Scala implementation. So we might want to try a different version number, in particular Scala Version 3, which has a completely new compiler (originally called Dotty). The Scastie web-based Scala environment might be handy for that (`https://scastie.scala-lang.org/`).

Since we know (from previous weeks) that `10 to 9 by 1` is an instance of the Scala library class `Range`, we could look up the source code for that library and see if it confirms our understanding; it does! (The code can be found by following a link in the Scala API documentation; it is at `https://github.com/scala/scala/blob/v2.12.12/src/library/scala/collection/immutable/Range.scala#L62`.)

2. (a) Follows immediately from a single rule, since `(Y=2)[X/Y]` is just `X=2`.

```
-----------------------------------------(ASSIGN)
      { X = 2 } Y := X { Y = 2 }
```

(b) Here we need to use the (CONSEQ) rule to make things "match up" before applying the (ASSIGN) rule.

```
    (arithmetic)
---------------------  -----------------------------------------(ASSIGN)
  X > 2 => X + 1 > 3         { X + 1 > 3 } X := X + 1 { X > 3 }
-------------------------------------------------------------(CONSEQ)
       { X > 2 } X := X + 1 { X > 3 }
```

Here I've explicitly shown the `P => P'` hypothesis to (CONSEQ) and justified it by an appeal to "arithmetic." In practice, we generally omit the `P => P'` and `Q' => Q` hypotheses in (CONSEQ) rules, since we can see what they must be by comparing the preconditions and postconditions below and above the line, and we implicitly appeal to "arithmetic and logic" to justify them.

(c) Again, we must use a (CONSEQ) node to match things up:

```
-----------------------------------(ASSIGN)
       { 3 = 3 } X := 3 { X = 3 }
-----------------------------------------(CONSEQ)
       { X = 2 } X := 3 { X = 3 }
```

Here the necessary "arithmetic and logic" is that `X = 2 => 3 = 3`, which is certainly true.

(d) This triple is not valid. In order to apply (ASSIGN), we would first need to derive

```
       { -X <= 0 } X := -X { X <= 0 }
-----------------------------------------(CONSEQ???)
        { X <= 0 } X := -X { X <= 0 }
```

But in general it is *not* the case that `X <= 0 => -X <= 0`. In fact, this is only true when `X = 0`. So this (CONSEQ) step is invalid.

(e) This is a straightforward use of the COMP rule. The only challenge is to "guess" the state in between the two statements, which is easy in this example.

```
-----------------------------------(ASSIGN)      -----------------------------------(ASSIGN)
 { X + 1 = 1 } X := X + 1 { X = 1 }                 { X + 1 = 2 } X := X + 1 { X = 2 }
-----------------------------------(CONSEQ)      -----------------------------------(CONSEQ)
 { X = 0 } X := X + 1 { X = 1 }                     { X = 1 } X := X + 1 { X = 2 }
-------------------------------------------------------------------------------------(COMP)
                     { X = 0 }  X := X + 1; X := X - 1 { X = 0 }
```

(f) Informally, any triple with `False` as its precondition is valid. Formally, this follows from the CONSEQ rule, where we have to justify that `False => 3 = 2`. In logic, `False` implies anything, so this is fine!

```
-------------------------------(ASSIGN)
 { 3 = 2 } X := 3 { X = 2 }
---------------------------------(CONSEQ)
 { False } X := 3 { X = 2 }
```

Another way to think about this is that a triple asserts what is true in the poststate *if* the precondition is true in the prestate. If the precondition is `False`, it is *never* true in the prestate, so the triple's assertion is vacuously true.

(g) This is a simple use of the `COND`itional rule. Note that the left-hand (CONSEQ) step is valid because `X = 2 /\ X > 0 => 3 = 3` (trivially) and the right-hand one is valid because `X = 2 /\ X < 0 = False` and `False => 4 = 3`.

```
-------------------------------(ASSIGN)        -------------------------------(ASSIGN)
   { 3 = 3 } Y := 3 { Y = 3 }                     { 4 = 3 } Y := 4  { Y = 3 }
-----------------------------------(CONSEQ)   -------------------------------------(CONSEQ)
 { X = 2 /\ X > 0 } Y := 3 { Y = 3 }             { X = 2 /\ X <= 0 } Y := 4 { Y = 3 }
-------------------------------------------------------------------------------------(COND)
             { X = 2 } if X > 0 then Y := 3 else Y := 4 endif { Y = 3 }
```

(h) This is a simple use of the (WHILE) rule, with the invariant `X >= 0`. Again, we have to get things lined up just right to match the form of the (WHILE) rule, in particular in the post-condition, which requires

us to check the strange-looking, but clearly true, fact that `(X >= 0) /\ not (X > 0) => X = 0` Notice that the invariant is serving an important purpose here: without it, the strongest postcondition we could prove for the whole statement would be `X <= 0`, rather than `X = 0`.

```
            ---------------------------------- (ARITH)
            {X - 1 >= 0} X := X - 1 {X >= 0}
          ----------------------------------------(CONSEQ)
           {X >= 0 /\ X > 0} X := X - 1 { X >= 0 }
-------------------------------------------------------------------- (WHILE)
{ X >= 0 } while (X > 0) do X := X - 1 { X >= 0 /\ not (X > 0) }
-------------------------------------------------------------------- (CONSEQ)
          { X = 10 } while (X > 0) do X := X - 1 { X = 0 }
```

3. The `after` statement is just sequential composition in reverse, so we can just take the (COMP) rule and switch the statements:

```
 {P} S2 {Q}   {Q} S1 {R}
--------------------------(AFTER)
   {P} S1 after S2 {R}
```

Then we can prove:

```
 ------------------------------------(ASSIGN)     ------------------------------------(ASSIGN)
   { X + 1 = 1 } X := X + 1 { X = 1 }                 { X * 2 = 2 } X := X * 2 { X = 2 }
------------------------------------(CONSEQ)      ------------------------------------(CONSEQ)
     { X = 0 } X := X + 1 { X = 1 }                     { X = 1 } X := X * 2 { X = 2 }
------------------------------------------------------------------------------------------ (AFTER)
               { X = 0 } X := X * 2 after X := X + 1 { X = 2 }
```