Exercise 1:

Conceptual question: In the latter example, how could we change the code without altering the postcondition? How does the "forgetting" of assertions correspond to a form of modularity?

Answer We can change it to anything such that `z > 1`, e.g.: increasing the value of `x` and `y`. Weakening assertions means that they describe a larger space of behaviors, so there are more ways to change the program and still have that assertion hold.

Exercise 2

Look at the code in Figure 1. Fill in all assertions.

Answer

```
  {  a = 0 }
1 { (2-a) * 2 + 1 = 5}
2 b := 2 -a
3 { b * 2 + 1 = 5 }
4 c := b * 2
5 { c+1 = 5 }
6 d := c + 1
7 { d = 5}
```

Exercise 3

We haven't given you any rules other than the ones for straight-line code. But we can still write some interesting programs if we use integer division, which we introduce in this exercise. In this exercise, assume all variables are integers, and all division is integer division (rounds toward 0)

```
1 { x > 0 }
1 { x = x }
1 { ( ( x is odd ) => x = x ) /\ ( ( x is even ) => x = x }
1 { ( ( x is odd ) => ( x / 2 ) = (x - 1) / 2 ) /\ ( ( x is even ) =>
 ↪  ( x / 2 ) = x / 2 ) } // integer division 1/2 = 0
1 { ( ( x is odd ) => ( x / 2 ) * 2 = x - 1 ) /\ ( ( x is even ) => (
 ↪  x / 2 ) * 2 = x ) }
2 y := ( x / 2 ) * 2
3 { ( ( x is odd ) => y = x - 1 ) /\ ( ( x is even ) => y = x ) }
3 { ( ( x is odd ) => x - y = 1 ) /\ ( ( x is even ) => x - y = 0 }
4 z := x - y
5 { ( ( x is odd ) => z = 1 ) /\ ( ( x is even ) => z = 0 ) }
5 { ( ( x is odd ) => z * 5 + ( 1 - z ) * 12 = 5 ) /\ ( ( x is even )
 ↪  => z * 5 + ( 1 - z ) * 12 = 12 ) }
```

```
6 a := z * 5 + ( 1 - z ) * 12
7 { ( ( x is odd ) => a = 5 ) /\ ( ( x is even ) => a = 12 ) }
```

Answer

Exercise 4

Question 4.1

This exercise is much harder, but is meant to illustrate two important points in software design: that conditionals are not simply "if-statements" and can appear even in straight-line code, and that Hoare logic gives an objective measure of complexity which can motivate refactoring. We will do part of this example now, and revisit in again in the next section. In this exercise, assume all variables are integers, all division is integer division (rounds toward 0), and that `x / 0 == 0` for all `x`.

```
1.  { true }
1.  { ((a <= 0) =>  1= 1) /\ ((a > 0) => 0 = 0) }
1.   { ((a <= 0) =>  (2-(a+1)/a)/2 = 1) /\ ((a > 0) =>  (2-(a+1)/a)/2
 ↪  = 0) }
2.   d := (2-(a+1)/a)/2
3.   { ((a <= 0) => d = 1) /\ ((a > 0) => d = 0) }
3.   { ((a <= 0) => d*2+(1-d)*3 = 2) /\ ((a > 0) => d*2+(1-d)*3 = 3) }
4.   m := d*2 + (1-d)*3
5.   { ((a <= 0) => m = 2) /\ ((a > 0) => m = 3) }
6.   x := b * 2
7.   { ((a <= 0) => m*x = 4*b) /\ ((a > 0) => m*x = 6 * b) }
8.   x := x * 2
9.   { ((a <= 0) => m*x = 8*b) /\ ((a > 0) => m*x = 12 * b) }
10.  x := m * x
11.  { ((a <= 0) => x = 8*b) /\ ((a > 0) => x = 12 * b)
12.  x := x + 1
13.  { ((a <= 0) => x = 8*b+1) /\ ((a > 0) => x = 12 * b + 1)
```

Answer **A common mistake**: Trying to guess a forward version of the assignment rule and using it instead (but trying to go off intuition, instead of using it strictly):

I say this is a mistake in that, while it produces valid pre/postconditions, it does not produce either the strong pre or post conditions on many lines, and doesn't achieve the goal of deriving these results mechanically rather than using human intuition.

Example: `Line 9`, having: `{( x = b*4 ) /\ ( (a <= 0) => m = 2 ) /\ ( (a > 0) => m =`

Response: " `x=b*4` " is a non-sequitur (does not follow logically) if you are going backwards. It's correct if you are going forwards (which I didn't give you the rules for), but, if you're going forward, you also need to keep a lot of other stuff around

The forward version of the assignment rule can be phrased as follows:

`{P, old_x = x} x := e {[old_x/x]P /\ x = [old_x/x]e}`

(The actual version is phrased with some extra machinery in a way that makes it a bit easier to track old values of variables.)

So, you would actually need to keep around extra information about the values of d and m. Otherwise, the postconditions you're getting are not the strongest ones; you should be able to add later code which references those variables, but your assertions won't let you.

Extra information for the curious:

Here's how to prove line 2 implies line 3:

`d := (2−(a+1)/a)/2;`

`x / y` can be defined as:

```
If y = 0,
    then 0
Else if  x and y are the same  sign,
    then the unique number n such that n|x| <= |y| < (n+1)|x|
Else if x and y are different signs,
   then -n, where n is the unique number n such that n|x| <= |y| < (n+1)|x|
```

These cases for division feed into the cases of the proof. There are 5 cases:

- The a>1 case (then a+1/a=1)
- The 1 case (then a+1/a=2)
- The 0 case
- The -1 case
- The a<-1 case (then a+1/a = 0).

Alternate proof:

```
Break the assignment into this:

{true}
temp1 := integerDivision(a + 1,a);
{A}
temp2 := 2 - temp1;
{B}
d := integerDivision(temp2, 2);
{C}


The assertions after each line are:


{A}
0 if  a<0
0 if  a=0 (given the condition that x/0 = 0).
2 if  a=1
1 if  a>0
This can be expressed as { (a <= 0) => temp1=0, (a=1) => temp1 = 2, (a>1) => temp1=0 }
```

```
{B}
2 if  a<=0
0 if  a=1
1 if  a>0
This can be expressed as { (a <= 0) => temp2=2,  (a=1) => temp2 = 0, (a>1) => temp2=1 }


{C}
1 if  a<=0
0 if  a>0
That can be expressed as { (a <= 0) => d = 1,  (a > 0) => d = 0  }
```

Question 4.2

In what sense does it contain a conditional?

Answer The postcondition is of the form `if a<= 0 then Q else R,` i.e.: it does something different based on whether `(a <= 0)` .

This happens because the statement `d := (2-(a+1)/a)/2` was designed to be equivalent to `d := (a <= 0) ? 1 : 0` , and the following line was designed to be equivalent to `if (d) m := 2 else m := 3` ". Using these techniques, I can take any arithmetic program with conditionals, and convert it into an equivalent straight-line program. I've seen times when people were asked to remove the conditional from a program, and they unknowingly did something like this. They remove the syntactic conditional, but not the logical conditional.

The magic expression is `d*<something> + (1-d)*<something else>` as a general template equivalent to `if d then <something> else <something else>` . Chaining these together can turn arbitrary branching control flow into straight-line code. In digital design, these kinds of expressions are called "multiplexers," and they play a critical role in constructing CPUs.

For another example, in a language with arrays, you could also turn `if (d == 9) m := 2 else m :=` into `m := [2, 3][d]` . If you write out what this line does as a logical formula, it works the same way.

Exercise 5

Revisit the code in Figure 3. Notice how it required you to track a lot of information between some of the intermediate lines. How might you be able to reorder the statements to make the code simpler? The precondition and postcondition of the code as a whole should remain the same.

Answer Move lines `6 and 8` to the top.

New derivation:

```
1.   { true }
2.   x := b * 2
3.   { x = 2 * b }
4.   x := x * 2
5.   { x = 4*b }
6.   d := (2-(a+1)/a)/2
7.   { ((a <= 0) => d = 1 /\ 2*x = 8*b) /\ ((a > 0) => d = 0 /\ 3*x =
  ↪  12 * b) }  // consequence rule
7.   { ((a <= 0) => (3-d)*x = 8*b) /\ ((a > 0) => (3-d)*x = 12 * b) }
8.   m := d*2 + (1-d)*3
9.   { ((a <= 0) => m*x = 8*b) /\ ((a > 0) => m*x = 12 * b) }
10.  x := m * x
11.  { ((a <= 0) => x = 8*b) /\ ((a > 0) => x = 12 * b)
12.  x := x + 1
13.  { ((a <= 0) => x = 8*b+1) /\ ((a > 0) => x = 12 * b + 1)
```

Exercise 6

Prove this sequential search procedure correct by choosing a proper loop invariant.

Answer Loop invariant: `(forall j, (j >= 0) && (j < i) => arr[j] != val)`

Full proof:

```
{ true }
i := 0
{ forall j, (j >= 0 && j < i) => arr[j] != val  }
while arr[i] != val && i < n do
   { i < n /\ arr[i] != val /\ (forall j, (j >= 0 && j < i) => arr[j] != val) }
    { i < n /\ (forall j, (j >= 0 && j <= i) => arr[j] != val) }
    i := i + 1
    { (i -1 < n) /\ forall j, (j >= 0 && j < i) => arr[j] != val) }
    { forall j, (j >= 0 && j < i) => arr[j] != val) }
end
{ arr[i] == val || ((forall j, (j >= 0 && j < n) => arr[j] != val) /\ i = n) }
```