Exercises 3

1 Loop semantics

Compute and simplify the relation corresponding to the following programs:

```
y = 0
while (z > 0) {
  y = y + x
  z = z − 1
}
```

```
c = 0
while (b >= a) {
  b = b − a
  c = c + 1
}
```

Solution

- $x' = x \land ((z \leq 0 \land z = z' \land y' = 0) \lor (z > 0 \land z' = 0 \land y' = x * z))$
- $(b < a \land a' = a \land b' = b \land c' = 0) \lor (b \geq a \land a' = a \land b' = b \bmod a \land c' = b/a)$

2 Loop invariants

In the following program, provide the necessary loop invariant for verification to succeed:

```
def binarySearch(a: Array[BigInt], key: BigInt): Int = ({
  require(a.length > 0 && forall { (i: Int, j: Int) =>
    (i >= 0 && j >= 0 && i < a.length && j < a.length && i < j) ==> (a(i) <= a(j))
  })

  var low = 0
  var high = a.length − 1
  var res = −1

  (while(low <= high && res == −1) {
    val o = if ((high & 1) == 1 && (low & 1) == 1) 1 else 0
    val i = high / 2 + low / 2 + o
    val v = a(i)

    if(v == key)
      res = i

    if(v > key)
      high = i − 1
    else if(v < key)
      low = i + 1
  }) invariant(TODO)
  res
}) ensuring(res => {
```

```
  if(res == -1)
    forall((i: Int) => (0 <= i && i < a.length) ==> (a(i) != key))
  else
    a(res) == key
})
```

The following invariant enables verification:

```
0 <= low && low <= high + 1 && high < a.length &&
(if(res == -1)
  forall((i: Int) => (0 <= i && i < low) ==> (a(i) != key)) &&
  forall((i: Int) => (high + 1 <= i && i < a.length) ==> (a(i) != key))
else
  res >= 0 && res < a.length && a(res) == key)
```

3 Proof construction

Show that `list flatMap f flatMap g == list flatMap (x => f(x) flatMap g)` using the following axioms:

1. `Nil flatMap f == Nil`
2. `(x :: xs) flatMap f == f(x) ++ (xs flatMap f)`
3. `Nil ++ xs == xs`
4. `xs ++ Nil == xs`
5. `(x :: xs) ++ ys == x :: (xs ++ ys)`

Use the proof strategies you have seen in Welder, such as structural induction and equational reasoning. Make sure each step in your reasoning is clearly indicated.

Hint: It may be useful to introduce some auxiliary lemmas.

Solution Let us start by defining the two following helper functions:

```
def lhs(glist, flist, list) = glist ++ ((flist ++ (list flatMap f)) flatMap g)
def rhs(glist, flist, list) = glist ++ (flist flatMap g) ++ (list flatMap (x => f(x) flatMap g))
```

We now show the more general lemma `lhs(glist, flist, list) == rhs(glist, flist, list)`.

Induction on list

We start by using structural induction on `list` to show that

`∀ flist, glist. lhs(glist, flist, list) == rhs(glist, flist, list)`

- Base case `list == Nil`:

```
lhs(glist, flist, Nil) == glist ++ ((flist ++ (Nil flatMap f)) flatMap g)
                       == glist ++ ((flist ++ Nil) flatMap g)
                       == glist ++ (flist flatMap g)
                       == glist ++ (flist flatMap g) ++ Nil
                       == glist ++ (flist flatMap g) ++ (Nil flatMap (x => f(x) flatMap g))
                       == rhs(glist, flist, Nil)
```

- Inductive case list == (x :: xs):

  Induction on flist

  We then use structural induction on flist to show that

  ∀ glist. lhs(glist, flist, x :: xs) == rhs(glist, flist, x :: xs)

  - Base case flist == Nil:

    ```
    lhs(glist, Nil, x :: xs) == glist ++ ((Nil ++ (x :: xs) flatMap f) flatMap g)
                             == glist ++ (((x :: xs) flatMap f) flatMap g)
                             == glist ++ ((f(x) ++ (xs flatMap f)) flatMap g) // IHS list
                             == glist ++ (f(x) flatMap g) ++ (xs flatMap (x => f(x) flatMap g))
                             == glist ++ (Nil flatMap g) ++ ((x :: xs) flatMap (x => f(x) flatMap g))
                             == rhs(glist, Nil, x :: xs)
    ```

  - Inductive case flist == (y :: ys):

    Induction on glist

    We finally use structural induction on glist to show that

    lhs(glist, y :: ys, x :: xs) == rhs(glist, y :: ys, x :: xs)

    * Base case glist == Nil:

      ```
      lhs(Nil, y :: ys, x :: xs) == Nil ++ ((y :: ys) ++ ((x :: xs) flatMap f)) flatMap g)
                                 == ((y :: ys) ++ ((x :: xs) flatMap f)) flatMap g
                                 == g(y) ++ ((ys ++ ((x :: xs) flatMap f)) flatMap g) // IHS flist
                                 == g(y) ++ (ys flatMap g) ++ ((x :: xs) flatMap (x => f(x) flatMap g))
                                 == ((y :: ys) flatMap g) ++ ((x :: xs) flatMap (x => f(x) flatMap g))
                                 == Nil ++ ((y :: ys) flatMap g) ++ ((x :: xs) flatMap (x => f(x) flatMap g))
                                 == rhs(Nil, y :: ys, x :: xs)
      ```

    * Inductive case glist = (z :: zs):

      ```
      lhs(z :: zs, y :: ys, x :: xs) == (z :: zs) ++ ((y :: ys) ++ ((x :: xs) flatMap f)) flatMap g)
                                     == z :: (zs ++ (((y :: ys) ++ ((x :: xs) flatMap f)) flatMap g)) // IHS glist
                                     == z :: (zs ++ ((y :: ys) flatMap f) ++ ((x :: xs) flatMap (x => f(x) flatMap g)))
                                     == (z :: zs) ++ ((y :: ys) flatMap f) ++ ((x :: xs) flatMap (x => f(x) flatMap g))
                                     == rhs(z :: zs, y :: ys, x :: xs)
      ```