IMPERATIVE PROGRAMMING HT2018

SHEET 2

GABRIEL MOISE

QUESTION 1

```
object Question1

{

  def swap (x : Int, y: Int) = {val t = x; x = y; y = t}

  // We will get an error here for reassigning x and y with new values. They have to stay the same because they are the arguments of the function so they behave in the same way as val variables do.

  // error: reassignment to val

  def swapEntries(a: Array[Int], i: Int, j: Int) = {

     val t = a(i); a(i) = a(j); a(j) = t

   }

  // Here we get no error as long as i and j do not exceed the length of the array. This is because although a is a value array, we can modify its entries (here we interchange entry i with entry j)

}
```

QUESTION 2

```
object SideEffects

{

   var x = 3; var y = 5

   def nasty (x: Int) : Int = { y = 1; 2 * x }

   def main (args: Array[String]) = println(nasty(x)+ y)

   // In this form, we first make y equal to 1 then we double x, so it becomes 6, then at printing we have 6 + 1 (the new value of y) from the "nasty" function. If we had println(y + nasty(x)), first we would have y = 5, then nasty (x), which returns 2*x = 6, so the integer returned would be 11. The order matters as we perform the addition from left to right, calculating each function we need before performing the next addition.

}
```

QUESTION 3

```
object Question3

{

  /* Function that two arrays are the same or not*/

  def equal (a: Array[String], b: Array[String]) : Boolean =

   {

     val n = a.size

     val m = b.size
```

```
  // Obviously, if the arrays are of different sizes, they cannot be equal
  if (n!=m) return false
  var i = 0
  // Invariant I: a[0..i) = b[0..i) && 0<=i<=n
  while (i<n)
  {
    // 0<=i<n
    if (a(i) != b(i)) return false
    // If we got past this point, then a(i) == b(i), so we have a[0..(i+1)) = b[0..(i+1))
    i += 1
    // I
  }
  // I holds, so a[0..n) = b[0..m) (as m=n), so the two arrays are identical
  true
}
// This function is an order that I defined:
// String s1 is smaller than s2 if s1 reversed is smaller than reversed s2
def ord (s1: String, s2: String) : Boolean =
  {
     // We turn the 2 strings into arrays
     val a = s1.toArray
     val b = s2.toArray
     // We reverse their elements
     val reva = a.reverse
     val revb = b.reverse
     // We turn them back into strings
     val revs1 = new String (reva)
     val revs2 = new String (revb)
     // We check if revs1 is less than or equal to revs2 and we return the answer
     revs1 <= revs2
  }
def main (args : Array[String]) =
  {
    val test1 = Array ("banana","apple","college","professor","dog","cat")
```

```
    val sort1 = test1.sorted
    val result1 = Array ("apple","banana","cat","college","dog","professor")
    assert (equal(sort1,result1))
    val test2 = Array ("aaaa","aa","aaa","aaaaaa","a","aaaaa")
    val sort2 = test2.sorted
    val result2 = Array ("a","aa","aaa","aaaa","aaaaa","aaaaaa")
    assert (equal(sort2,result2))
    val order : ((String,String) => Boolean) = (s1,s2) => {ord(s1,s2)}
    val test3 = Array ("banana","apple","college","professor","dog","cat")
    val sort3 = test3.sortWith(order)
    val result3 = Array ("banana","college","apple","dog","professor","cat")
    assert (equal(sort3,result3))
    val test4 = Array ("a","a","a","a","a","a","a")
    val sort4 = test4.sortWith(order)
    val result4 = Array ("a","a","a","a","a","a","a")
    assert (equal(sort4,result4))
  }
}
```

QUESTION 4

```
object Question4
{
 def main (args: Array[String]) =
 {
  val timeEnd:Double = 1.0
  val numSteps:Int =
  val timeStep:Double = timeEnd/numSteps
  // timeEnd=numSteps*timeStep and numSteps∈N
  var time = 0.0
  while (time < timeEnd-1e-10)
  {
   // Inv: 0 <= time <= timeEnd and time=k*timeStep for some k∈N
   time += timeStep
  }
  // Inv => time == timeEnd
```

```
  }
}
```

/*(a) After each iteration of the loop, time is in an interval of the form (n*timeStep-error,n*timeStep+error), where n is the number of iterations done so far and error is a very small value than is obtained from the fact that the arithmetic we use on floating points is not precise because at each step we can have an error of epsilon, which is undetectable by the machine. So, after numSteps operations, time is in (timeEnd-error,timeEnd+error). If we are in the first half of the interval, we need an additional iteration of the loop for time to exceed timeEnd, so (numSteps+1) iterations, and if not, then we are done and we needed just numSteps iterations. (Notice that the error is still very small compared to timeStep).

(b) At the end of the loop, if we were in the case when we needed numSteps iterations, time will be in the interval [timeEnd,timeEnd+error) so we are very close to the value we want. However, if we needed numSteps+1 iterations, then time was in the interval (timeEnd-error,timeEnd), so by adding a timeStep, we will be in the interval (timeEnd-error2+timeStep,timeEnd+timeStep+smallError), so we can be at a distance of approximately timeStep far from the result we wanted. (error2 is a little bigger than error and smallError is a very small error compared to error or error2)

(c) Let's say that instead of time<timeEnd in the guard of the while we would have time<limit. Our goal is to find limit such that time will be as close as possible to timeEnd at the end of the loop even in the worst case scenario. Here, we can use the fact that after n iterations, time is in (n*timeStep-error,n*timeStep+error). By setting limit to be timeEnd-error, then we know that after numSteps iterations time will be greater than the limit and in the worst case scenario, it will be timeEnd+error, which is greater than the limit by 2*error, which is pretty close. I tried error to be 1e-10, therefore limit= (timeEnd - 1e-10) and it worked on numbers up to 9 digits.*/

QUESTION 5

```
object Question5
{
  /** Does pat appear as a substring of line?
  * i.e. pat [0..K) = line [i..i+K) for some i in [0..j). */
  def search (pat: Array[Char], line: Array[Char]) : Boolean =
    {
      val K = pat.size ; val N = line.size
      // Invariant I: found = (line[i..i+K) = pat [0..K) for some i in [0..j)) and 0<=j<=N-K
      var j = 0 ; var found = false
      while (j <= N-K && !found)
      {
        // set found if line [j..j+K) = pat [0..K)
        // Invariant : line [j..j+k) = pat [0..k)
        var k = 1
        while (k<K && line(j+k)==pat(k)) k = k + 1
        found = (k==K)
        j = j + 1
      }
      // I && (j=N-K+1 || found)
```

```
    // found = (line[i..i+K) = pat [0..K) for some i in [0..N-K+1) )
    found
  }
 def main (args: Array[String]) =
  {
    // (a) If we set found to be true initially, then we won't enter the while loop, so search will always return true for any input. So, we will get an error for any false case, like this one:
    val pat1 = Array ('A') ; val line1 = Array ('B')
    assert (search(pat1,line1) == false)
    // (b) Here, if we replace the <= in the first while condition with <, we will not consider the possibility that pat is the suffix of line, because the suffix of K characters is line[N-K..N), but in order to verify if pat2 is equal to that we would need j to reach N-K inclusively
    val pat2 = Array ('a') ; val line2 = Array ('c','b','a')
    assert (search (pat2,line2) == true)
    // (c) If we replace N-K with N-K+1 in the first while condition we will get ArrayIndexOutOfBoundsException because in the case when j = N-K+1, in the second while when k reaches K-1 we will test if line(j+k), meaning line(N) which is where we get out of the array, because line has just N elements, indexed from 0 to N-1 inclusively.
    // (d) This test fails because when we check with the second while if line [j..j+K) = pat [0..K) we start from k = 1 to K-1 and we do not check for the first character. Also, as we can see the Invariant doesn't hold initially before getting into the while loop: line [j..j+k) = pat [0..k) is no longer true if we start with k=1, as line(j) might differ from pat(0).
    val pat3 = Array ('b','o','a','t') ; val line3 = Array ('R','a','i','n','c','o','a','t')
    assert (search (pat3,line3) == false)
    // (e) We get ArrayIndexOutOfBoundsException again because when we get to k = K, pat(k) will be in fact pat(K), which doesn't exist because we got out of the array.
    // (f) If we say found = (k>=K) this won't affect the program in any way because k at each step in the second while loop can increase only by 1, so at the end of the loop, if we found pat in line we will have k=K so found will be true whether we set the condition to be k==K or k>=K, and for k<K we will get false anyways.
  }
}
```

QUESTION 6

```
object Question6
{
 /** For a given value n, check if after the first n characters, s starts repeating itself*/
 def equal (n: Int, N: Int, s: Array[Char]) : Boolean =
  {
    var i = 0
    // Invariant s[0..i)=s[n..i+n) && 0<=i<=N-n
    // Variant N-n-i
```

```
    while (i<N-n)
    {
      if (s(i) != s(i+n)) return false else i += 1
      // If we find that two characters differ then we return false, else we continue by increasing i by 1, so in the loop, if we don't get out by returning false, the invariant holds
    }
    // At the end, if we haven't returned false, then it means that we have i=N-n and from the invariant we have that s[0..N-n) = s[n..N), so we return true
    true
  }
  /** Calculating the number of characters in s after which it starts repeating itself*/
 def search (s : Array[Char]) : Int =
  {
    val N = s.size
    var n = 1
    // Invariant : the period of s is greater or equal to n
    // variant : N-n
    while (n<=N)
      if (equal (n,N,s)) return n else n +=1
      // If after the first n characters s starts repeating itself, then we return n, else we increase n and we continue checking
    0
    // We will never return 0 because for n=N, equal(n,N,s) will always be true, so the result is always less than or equal to N
  }
 def main (args: Array[String]) =
  {
    val test1 = Array ('a','a','a','a')
    assert(search(test1) == 1)
    val test2 = Array ('a','b','c','d','e')
    assert(search(test2) == 5)
    val test3 = Array ('a')
    assert(search(test3) == 1)
    val test4 = Array ('a','b','a','b','a','b','a','b')
    assert(search(test4) == 2)
    val test5 = Array ('a','b','c','d','a','b')
    assert(search(test5) == 4)
```

```
  }
}
```

QUESTION 7

```
object Question7
{
 /** Testing if there exists an i in [0..N) that satisfies p*/
 def exists(p : Int => Boolean, N : Int): Boolean =
  {
   var i = 0
   // Invariant I : there is no element in [0..i) that satisfies p && 0<=i<=N
   // variant N-i
   while (i < N)
   {
    // I && 0<=i<N
    if (p(i)) return true
    // If we returned true then we found a value that satisfies p. If not, then p(i) didn't hold, so p(i) = false
    // I && p(i) false => there is no element in [0..(i+1)) that satisfies p && 0<=i<N
    i += 1
    // I : there is no element in [0..i) that satisfies p && 0<=i<=N
   }
   // If we didn't return true yet, because I holds we have no element in [0..N) that satisfies p, so we return false
   false
  }
 def main (args: Array[String]) =
  {
   val prop1: (Int => Boolean) = i => {i%5==0}; val n1 = 20
   assert (exists(prop1,n1) == true)
   val prop2: (Int => Boolean) = i => {i%5==0}; val n2 = 0
   assert (exists(prop2,n2) == false)
   val prop3: (Int => Boolean) = i => {i%11==0 && i!=0}; val n3 = 10
   assert (exists(prop3,n3) == false)
   val prop4: (Int => Boolean) = i => {i*i>=400}; val n4 = 21
   assert (exists(prop4,n4) == true)
   val prop5: (Int => Boolean) = i => {i*i>=400}; val n5 = 20
```

```
    assert (exists(prop5,n5) == false)
  }
}
```

QUESTION 8

```
object Question8
{
 // (a) As p/q >= 1/m and 1/m is the biggest reciprocal with this property, then m is the smallest integer with this property, we then have m>=q/p, so in order for m to be the smallest with this property, we need m=ceiling(q/p). (Here / is not div, but the fraction sign)
 def ceiling (p : Int, q : Int) : Int =
  {
    // m>=q/p if and only if m*p>=q (we are guaranteed that 0<p<q)
   //m is non-negative as 0<p<q and non-zero because we can't divide by 0
    var m = 1
    // We search for m by incrementing it by 1 and testing if we exceeded q/p. When we do, we stop, so m is the smallest integer with m>=q/p, then m=ceiling(q/p) as we needed
    while (m*p<q) m += 1
    m
  }
 /** Decomposing P/Q into a sum of reciprocals*/
 def sum (P : Int, Q : Int) : Array[Int] =
  {
    var p = P ; var q = Q
    var n = 10000
    var a = new Array [Int] (n) // We assume that n is sufficiently large for all the reciprocals we want to calculate
    var k = 0 // The current number of reciprocals we have found to be part of the sum
    var m = 1
    // Once we found a reciprocal, we continue from the m we found, we do not try again from 1 because p/q - 1/m becomes smaller so the next m we will find will be greater than or equal to the previous one (we'll discuss the "equal to" part later)
    // Invariant I: p/q+sum(1/a(i))=P/Q with i in [0..k)
    // variant p (we'll prove that it decreases after every iteration of the loop)
    while (p!=0)
    {
      // We find the reciprocal with minimum m
      while (m*p<q) m += 1
```

```
      // We found a reciprocal so we put it in the array
      a(k) = m ; k += 1
      // Now, we will update p and q with p1 and q1 such that p1/q1 = p/q - 1/m = (p*m-q)/(q*m)
      val p1 = p*m - q
      // (c) We are assured that p1<p because p*m-q<p <=> p*(m-1)<q <=> (m-1)<q/p and we chose m to be the smallest integer with m*p>=q, or m>=q/p, therefore (m-1)<p/q, therefore p1<p so p decreases every time. Also p/q cannot get negative since we never subtract from p/q something bigger than it, so p>=0 and q>0 (q remains positive so p must also remain at least zero) therefore p will eventually reach zero
      val q1 = q*m // proof that q is always positive as we claimed
      p = p1 ; q = q1
      // The invariant I holds since the new fraction p/q is smaller with 1/m than the previous one and the sum of the elements of the array (whose size increased by 1) is greater with 1/m
    }
    // p becomes 0, so we get 0/q+the sum of the first k elements from a = P/Q, so we return the array which contains the first k elements of a (as those are the ones we are interested in, the rest are only zeros)
    val result = new Array [Int] (k)
    var i = 0
    while (i<k) {result(i) = a(i); i+=1}
    result
    // (d) Let's suppose that there are two values in result that are equal (from our algorithm they have to be on consecutive positions as m is at least as big as the previous one). Then, at some point we subtracted 1/m from p/q and then again 1/m from (p/q-1/m). That means that we could have subtracted 2/m from p/q, thus we could have subtracted 1/(ceiling(m/2)) from p/q, as ceiling(m/2)>=m/2. Then, the while loop would have stopped at ceiling (m/2) and add that to the result instead, thus we can't have two equal elements in the result array.
  }
  /* Some tests:
  scala> Question8.sum(25,26)
  res1: Array[Int] = Array(2, 3, 8, 312)
  scala> Question8.sum(1,26)
  res2: Array[Int] = Array(26)
  scala> Question8.sum(48,96)
  res3: Array[Int] = Array(2)
  scala> Question8.sum(3,8)
  res4: Array[Int] = Array(3, 24)
  */
}
```

QUESTION 9

```
object Question9
{
  def log3 (x: Double) : Int =
    {
      var y = 0
      // log3 of x with x>=1 is guaranteed to be at least 0, so the floor is also at least 0, so y is a non-negative integer
      var p = 1.00
      // Invariant: p = 3^y
      // variant x-p/3
      while (p<=x)
      {
        p = p * 3
        // p = 3^(y+1)
        y += 1
        // I
      }
      // We got out of the loop, so p>x, and because at the previous step we were in the loop, then p/3 is the biggest power of 3 which is less than or equal to x. Because p = 3^y from the invariant, p/3=3^(y-1), so (y-1) is the floor of log3(x), so we return (y-1)
      (y-1)
    }
  def main (args : Array[String]) =
    {
      assert(log3 (1) == 0)
      assert(log3 (2) == 0)
      assert(log3 (3) == 1)
      assert(log3 (26) == 2)
      assert(log3 (27) == 3)
      assert(log3 (1000) == 6)
    }
}
```

QUESTION 10

```
object Question10
{
 /** Calculating the value of the polynomial with coefficients in the array a, at x */
 def eval(a: Array[Double], x: Double) : Double =
  {
   val n = a.size
   var i = 0
   var powx = 1.00
   var result = 0.00
   // Invariant I: result = the sum of the first i terms of the polynomial && powx = x^i && 0<=i<=n
   // variant (n-i)
   while (i<n)
   {
    // I && 0<=i<n
    result += powx * a(i)
    // result = the sum of the first (i+1) terms of the polynomial && powx = x^i
    i += 1
    // result = the sum of the first i terms of the polynomial && powx = x^(i-1) && 0<=i<=n
    powx = powx * x
    // I
   }
   // i=n so result = the sum of the first n terms of the polynomial
   result
  }
  // Notice that we do 2*n multiplications with this method
  def evalHarder(a: Array[Double], x: Double) : Double =
   {
    // Here we do only (n-1) multiplications to determine the value of the polynomial in x
    val n = a.size
    var i = 0
    var result = a(n-1) // = a(n-1) * x^0
    // Invariant I: result = sum(a(n-1-j)*x^(i-j)) with j from 0 to i && 0<=i<=n
```

```
    // variant (n-1-i)
    while (i<n-1)
    {
     // I && 0<=i<n
     result = result * x
     // result = sum (a(n-1-j)*x^(i-j+1)) with j from 0 to i, equivalent to
     // result = sum (a(n-k)*x^(i-k+2)) with k from 1 to (i+1), by setting k=j+1
     i += 1
     // result = sum (a(n-k)*x^(i-k+1)) with k from 1 to i, equivalent to
     // result = sum (a(n-1-j)*x^(i-j)) with j from 0 to (i-1), by setting j=k-1
     result = result + a(n-1-i)//*x^0
     // I
    }
    // i=n-1, from I we have result = sum(a(n-1-j)*x^(n-1-j)) with j from 0 to (n-1) and that's the result we needed
    // We made (n-1) multiplications, one for each iteration of the loop
    result
   }
}
```