| galois |

Solutions to the exercises

Exercise 1:

Both the number and its position are to be output. Therefore, state should be represented by a tuple (number,position) of two integers. The value of number will be the maximum value in lst and position will be its earliest position from the left. The comprehension will be assigned to variable res. Use i <- [0...] to get positions of numbers in lst as they are considered. Use p <- lst to get numbers from lst. Use ss <- res to get current state. State gets updated if ss.0 (current maximum) is less than p. The updated state is (p, i). A reasonable solution is:

```
fmax lst = res ! 0
  where
    res = [(-1,-1)]#[ if (ss.0 < p) then (p,i) else ss
                    | p <- lst | ss <- res | i <- [0...]]
```

Load from file and try it with the following:

```
fmax []                     // Returns (-1,-1) which is reasonable
fmax [4,7,2,3,7,1,0,6,5]    // Only non-negative numbers should be used
fmax [0,1,2,3,4,5,6,7,8]
```

Exercise 2:

State is a single Bit and is initially False. Let res be the value of the comprehension. Variable res is a sequence of Bits, the last of which is output using (res ! 0). Let ss <- res be the latest Bit of res (that is, the current State), let p <- lst be the next element of the input sequence lst. Let n be the number that is being checked for membership in lst. State becomes True and remains True if n is found to be a member of lst using (p == n), otherwise State remains False. A reasonable comprehension is the following:

```
member lst n = (res ! 0)
  where
    res = [False]#[ ss \/ (p == n) | p <- lst | ss <- res ]
```

Load from file and try it with the following:

```
member [] 10                   // Returns False
member [12,55,2,17,88,20] 17   // Returns True
member [12,55,2,17,88,20] 13   // Returns False
member [5,5,2,3,6,7] 5         // Returns True
member [5,5,2,3,6,7] 7         // Returns True
```

All numbers must be non-negative in this version

Exercise 3:

Call the value of the comprehension res. State is a tuple: the left side is a number from lst, the right side is a Bit where False means n has not yet been removed from lst and True means it has. The right side of State is used to ensure at most one occurrence of n is removed from lst. Current State q is obtained from q <- res. To remove n two sequences are maintained, namely [-1]#lst and lst#[-1]. The comprehension begins by choosing numbers za from [-1]#lst. As long as n is not found (checked with ~(za == n)), and q.1 is False, numbers from za are placed in the left side of the next State like this: (za, q.1). If n is found and q.1 is False then numbers from zb <- lst#[-1] are placed in the left side of State with the right side of State taking the value True (henceforth numbers are always taken from zb). The output needs to be only the sequence of numbers on the left side of State. This can be obtained from the comprehension [w.0 | w <- res]. The initial State is (-1, False). The first State added to res is also (-1, False) due to za=[-1]#lst. Therefore, the first two States of res are both (-1,False) and w always begins with two -1s. These can be removed with drop `{2} [w.0 | w <- res].

```
remove lst n = drop `{2} [w.0 | w <- res]
  where
    res = [(-1,False)]#[ if (q.1 == False /\ ~(za == n)) then (za, q.1) else (zb, True)
                       | za <- [-1]#lst | zb <- lst#[-1] | q <- res ]
```

Try it with the following:

```
remove [] 10                    // returns []
remove [12,55,2,17,88,20] 17    // returns [12,55,2,88,20,-1]
remove [12,55,2,17,88,20] 13    // returns [12,55,2,17,88,20]
remove [5,5,2,3,6,7] 5          // returns [5,2,3,6,7,-1]
remove [5,5,2,3,6,7] 7          // returns [5,5,2,3,6,-1]
remove [5,5,2,0,6,7] 0          // returns [5,5,2,6,7,-1]
```

Notes: without the -1 appended to the output sequence some output sequences would be shorter than others and this constitutes a type mismatch in Cryptol. Hence, the appended -1 when a number is removed from the input sequence is necessary. This will actually be exploited in the next Exercise. The above was designed for non-negative integer sequences.

Exercise 4:

Numbers x and y are to be tested for being permutations of each other. The strategy is as follows: consider numbers p from x iteratively (as p <- x). Let q be a tuple containing a subsequence of y on the left and a flag (Bit) on the right. Initially, q.0 is sequence y and q.1 is True (as q <- [y,True]#res where res is the value of the comprehension holding computation States). From iteration to iteration q.0 may be modified with numbers of y removed and -1s appended to it. This happens if p (from p <- x) is a member of the modified q.0 and then p is removed from q.0 and -1 appended to q.0 in that case. If all the numbers p, taken from x, remove all the numbers in q.0 then q.0 will contain all -1s when computation finishes and q.1 will have the value True as it will not be changed during the computation. If at some iteration p is not a member of q.0 then x can not be a permutation of y and the flag q.1 is set to False due to (member q.0 p) /\ q.1. having value False. Once the flag is set to False, it remains False for future iterations. When computation is complete the only concern is the flag which states whether the input sequences are permutations. The flag is removed from the right side of the last State with this: (res ! 0).1. A few extra lines are added to take care of two cases: both inputs are empty sequences, and both inputs do not have the same number of elements.

```
perm x y = if (length x) == 0 /\ (length y) == 0 then True else
           if (length x) == (length y) then (res ! 0).1 else False
  where
    res = [ ((remove q.0 p), (member q.0 p) /\ q.1) | q <- [(y,True)]#res | p <- x ]
```

Try it with the following:

```
perm [] []                    // returns True
perm [] [1]                   // returns False
perm [1] []                   // returns False
perm [1] [0]                  // returns False
perm [1] [1]                  // returns True
perm [1,1,2] [2,1,2]          // returns False
perm [1,2,3] [1,2,3,3]        // returns False
perm [3,2,1] [1,3,2]          // returns True
perm [1,2,3,4] [1,2,3]        // returns False
perm [1,2,3,3] [1,2,3]        // returns False
perm [11,44,23,82,66,91,42,41,63,72,82,12,16,21,91,72,71]  // returns True
     [44,91,72,11,71,42,12,63,16,23,91,82,21,72,41,82,66]  // note: two 91s,82s,72s
perm [11,44,23,82,66,91,42,41,63,72,82,12,16,21,91,72,71]  // returns False
     [44,71,72,11,71,42,12,63,16,23,91,82,21,72,41,82,66]
perm [97] ['a']               // returns True
```

Exercise 5:

State is a Bit which is `True` as long as the numbers `j1` seen up to now (from `j1 <- lst`) are in a non-decreasing sequence. State is initialized to `True`. Two sequences are maintained from the input sequence `lst`. One is `lst` itself (from `j1 <- lst`), the other is `lst` but ahead by 1 position (from `j2 <- drop `{1} lst`). Hence, `j1` and `j2` are two consecutive numbers in `lst`. If `j1`, the one in the lowest position of the two, is strictly greater than `j2`, in the next lowest position, then State becomes `False` and remains `False` until the end. To find the value of State at the end use `(res ! 0)`.

```
   nondecreasing lst = (res ! 0)
     where
       res = [True]#[ if (j1 <= j2) /\ k then True else False
                    | k <- res | j1 <- lst | j2 <- drop `{1} lst]
```

Try it on the following:

```
nondecreasing [2]                       // Returns True
nondecreasing [2,1]                     // Returns False
nondecreasing [1,2]                     // Returns True
nondecreasing [1,1,1]                   // Returns True
nondecreasing [1,2,1]                   // Returns False
nondecreasing [1,1,1,3,3,3,6,6,6,7]     // Returns True
```