

# Creating a puzzle Rotate

Helga Rykov Ibsen

November 18, 2021

## 1 Preface

The report at hand is part of the assignment 7g in Programming og Problemløsning, in which the students were asked to write a program for a game called Rotate. The task was formulated by the course coordinators Jon Sparring and Martin Elsmann.

## 2 Introduction

The report is based on the assignment 7g, in which we were asked to write a program for a puzzle called Rotate. The main idea about the game is somewhat similar to the well-known Rubik's Cube, where the puzzle can be solved through multiple rotations of the squares of the cube. In the case at hand, we are dealing with a square chess-like board of  $n \times n$ ,  $n \in \{2, 3, 4, 5\}$  fields, each having a unique position and is associated with a unique letter of the English alphabet from 'a' to 'z'.

For the puzzle to be solved, the board must obtain the solved configuration, which is the arrangement of letters in alphabetical order. Consider an illustration of the solved configuration of Rotate in Figure 1 below:

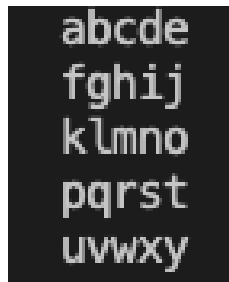


Figure 1: A solved configuration of a 5 x 5 board.

## 2.1 Problem statement

The task of the assignment is to build a program that generates rotate-puzzles. For each iteration, it should take a position from the player and apply the corresponding rotation until the puzzle is solved. The program should follow such requirements as:

- The program must use lists and not arrays.
- If the program includes loops, the loops must be programmed using recursion.
- The program is not allowed to use mutable values (or variables).
- The solution must be parameterized by  $n$ , the size of the board.
- The board must be represented as a list of letters. Thus, for  $n = 4$ , the board for a solved puzzle must be identical to the list ['a' .. 'p'].
- The program must consist of the following files:

```
game.fsx, rotate.fsi, rotate.fs, whiteboxtest.fsx,  
blackbostest.fsx
```

## 3 Description and evaluation of the program

In this assignment, we were asked to take the point of departure in writing the interface file that would include two user-defined types, named `Board` and `Position`, represented as a list of characters and an integer, respectively. It should also include the following functions:

```
create : n:uint → Board  
board2Str : Board → string  
validRotation : b:Board → p:Position → bool  
rotate : b:Board → p:Position → Board  
scramble : b:Board → m:uint → Board  
solved : b:Board → bool
```

Due to page restrictions of this paper, I will confine myself to discussing the design process of such functions as `create`, `validRotation`, `rotate` and `scramble`. The choice of the given functions is predicated on the idea that they make up the gist of the game Rotate.

### 3.1 create

The first function `create` has to take an integer `n` and return a `n x n` board. For instance, if `n = 3`, then the length of the board should be 9 (`3 x 3`).

Thus, I start out introducing a function that takes a board as the argument and return its length:

```
(* Given a board, return its length *)
let boardlength (Board list) : int =
  uint list.Length
```

Since our board should consist of the list of characters, we need to introduce another variable `alphabet` that would allow us to manipulate the otherwise non-deconstructable sum-type `Board`. My list of characters consists of the 25 letters of the English alphabet from 'a' to 'y'. The last letter 'z' has been left out due to the max `n = 5` (`5 x 5 = 25`).

```
let alphabet = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j';
                'k'; 'l'; 'm'; 'n'; 'o'; 'p'; 'q'; 'r'; 's'; 't';
                'u'; 'v'; 'w'; 'x'; 'y']
```

Now, the board for the game should look like a square of `n x n` letters and not a long list. For that purpose, we need to be able to split up the list into shorter sub-lists, each containing `n` elements. In other words, we need to introduce an additional function that would allow us to go iteratively through all the elements of the list `alphabet` and split it up into the slices of the length `n`:

```
(* Function to slice a list to a max of the n first items *)
let slice (list: 'a list) (n: int) =
  let rec r (l: 'a list) (acc: 'a list) =
    match l with
    | _ when acc.Length = n -> acc (* The acc length has the
      right length, so we should return it *)
    | [] -> acc (* The original list has no more items, so we
      should return what we have *)
    | head :: tail -> r tail (head :: acc) (* Add the next
      item *)
```

```
    List.rev (r list [] ) (* The 'r' function will reverse it, so
      we reverse it again before returning *)
```

Now that we can determine the length of our board as well as split it up into lists that are `n`-long, we can proceed to writing and testing the first function `create`.

Given the board of size `1 x 1` cannot be rotated and given there are only 26 letters in the English alphabet, the interval for `n` was set to be `[2; 5]`.

```
create (n:uint) =
  match n with
```

```

| _ when n < 2u -> Board [] (* A 1x1 board is not acceptable
    *)
| _ when n > 5u -> Board [] (* 5x5 is max, only 26 letters in
    alphabet *)
| _ -> Board (slice alphabet (int (n * n))) (* Return the
    board, based on a slice of the alphabet, n times n in size
    *)

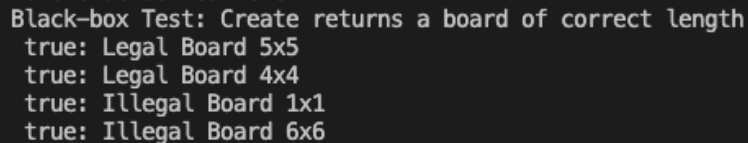
```

The black-box test showed that the function above returns as expected. Consider the print-statements and their output in the console in Figure 2 below (for the white-box tests, we refer to the document `whiteboxtest.fsx`):

```

(* Create returns a board of correct length *)
printfn "%5b: Legal Board 5x5" (boardlength(create 5u) = 25u)
printfn "%5b: Legal Board 4x4" ((boardlength (create 4u) = 16u))
printfn "%5b: Illegal Board 1x1" ((boardlength (create 1u) = 0u))
printfn "%5b: Illegal Board 6x6" (boardlength (create 6u) = 0u)

```



```

Black-box Test: Create returns a board of correct length
true: Legal Board 5x5
true: Legal Board 4x4
true: Illegal Board 1x1
true: Illegal Board 6x6

```

Figure 2: The result of the black-box test for `create`

### 3.2 validRotation

The function `validRotation` has to take a board and a rotation position as its arguments and return `true` or `false` depending on whether the position is valid or not.

This function should ensure that the rotation of letters can take place only if the following two conditions have been met: (1) none of the right-most column positions are valid inputs and (2) none of the bottom-most row positions are valid inputs to the rotation operation.

That means that we have to define the confines for valid rotation positions of the board. That is, we need to single out the last element in each row as well as the last row.

We start out introducing an additional function that would take a board as its argument and return the value `n` that it is made of. That will allow us to apply mathematical operations of modulo and subtraction for the conditions of the function `validRotation`:

```

let board2n (b: Board): int =
    int (System.Math.Sqrt(float (boardlength b)))

```

To single out the last element in each row, we employ the modulo operation. For example, for a board consisting of four rows and four columns, the last position in each row can be determined in the following way:

$$\begin{aligned}(4 \bmod 4) &= 0 \\(8 \bmod 4) &= 0 \\(12 \bmod 4) &= 0 \\(16 \bmod 4) &= 0\end{aligned}$$

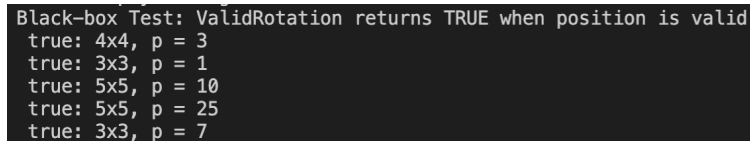
Now, we can proceed to writing and testing `validRotation`:

```
let validRotation (b: Board) (p: Position): bool =
    let n = uint (board2n b)

    match p with
    | Position v -> (v % n) <> 0u && v <= ((boardlength b) - n) (*
        Position is illegal in last column (every nth) and last
        row (length - n) *)
```

Consider the print-statements and their output in the console in Figure 4 below (for the white-box tests, we refer to the file `whiteboxtest.fsx`):

```
(* validRotation returns true when position is valid *)
printfn "%5b: 4x4, p=3" (validRotation(create 4u)(Position 3u)
    = true)
printfn "%5b: 3x3, p=1" (validRotation(create 3u)(Position 1u)
    = true)
printfn "%5b: 5x5, p=10" (validRotation(create 5u)(Position 10u)
    = false)
printfn "%5b: 5x5, p=25" (validRotation(create 5u)(Position 25u)
    = false)
printfn "%5b: 3x3, p=7" (validRotation(create 3u)(Position 7u)
    = false)
```



```
Black-box Test: ValidRotation returns TRUE when position is valid
true: 4x4, p = 3
true: 3x3, p = 1
true: 5x5, p = 10
true: 5x5, p = 25
true: 3x3, p = 7
```

Figure 3: The result of the black-box test for `validRotation`

### 3.3 rotate

The function `rotate` should take a board and an unsigned int  $m$  as its arguments and return a board identical to the original one, but where a local  $2 \times 2$  rotation

has been carried out at the indicated position. If the position is invalid, the function must return the original board.

The main component of the game logic should be a loop that keeps running while the `solved` function returns `false`. For each iteration, it should take a position from the player, and apply the corresponding rotation, if it is valid.

In our case, though, we are dealing with a list and need to implement the loop as recursion that would iterate through all the elements of the list and perform the rotation at a given position, if valid. And since we reference each element of a list within it by its index, we need to transform input positions of the list [1; 2; ... 25] into indexes [0; 1; 24] and perform rotation operations on the latter. A possible version of such function is presented below:

```
let rotate (b: Board) (p:Position): Board =
    let n = board2n b (* Get n *)
    let (Position uPosition) = p (* Get the position as an uint *)
    let posAsIndex = (int uPosition) - 1 (* Convert the 1-based
        position uint to a 0-based int index *)

    let (Board chars) = b (* Deconstruct the Board, so chars =
        char list *)

    let rec r (index: int) (acc: char list) = (* Function to
        iterate through all indexes of the list *)

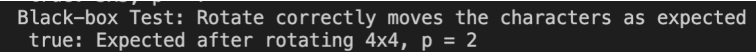
        match index with
        | i when i >= chars.Length -> acc (* When index has
            reached the list length, we are done *)
        | i when i = posAsIndex -> r (index + 1) ((chars.Item (
            index + n)) :: acc) (* If index = p, add item at index
            + n instead *)
        | i when i = posAsIndex + 1 -> r (index + 1) ((chars.Item
            (index - 1)) :: acc) (* If index = p + 1, add item at
            index - 1 instead *)
        | i when i = posAsIndex + n -> r (index + 1) ((chars.Item
            (index + 1)) :: acc) (* If index = p + n, add item at
            index + 1 instead *)
        | i when i = posAsIndex + n + 1 -> r (index + 1) ((chars.
            Item (index - n)) :: acc) (* If index = p + n + 1, add
            item at index - n instead *)
        | _ -> r (index + 1) ((chars.Item index) :: acc) (* We are
            outside the rotation area, add the item as is at
            index *)

    let result = List.rev (r 0 []) (* The result was reversed
        during add, reverse it again *)
    Board result (* Return as new board *)
```

For the purposes at hand, it will suffice to mention that the black-box tests revealed that `rotate` functions as expected. I will therefore confine myself to a single illustration of the print-statement and the output in the console in Figure 4 (for the extended black-box test of `rotate` on various board sizes and rotation positions, consider the file `blackboxtest.fsx`).

```
(* Rotate correctly moves the characters as expected *)
let board = create 4u
let expected = ['a'; 'f'; 'b'; 'd'; 'e'; 'g'; 'c'; 'h';
               'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o'; 'p']
let rotated = rotate board (Position 2u)
let result (Board x) = x

printfn "%5b: Expected after rotating 4x4, p=2" (result rotated
= expected)
```



```
Black-box Test: Rotate correctly moves the characters as expected
true: Expected after rotating 4x4, p = 2
```

Figure 4: The result of the black-box test for `rotate`

### 3.4 scramble

The last function to be discussed in this paper is `scramble` that takes a board and an unsigned int  $m$  as its arguments and returns another board, where all the elements of the original board have been rotated by  $m$  random legal rotations using `rotate`.

The core element of `scramble` is *random* rotation of the board  $m$  times before a new game. We can get a random built-in object by using `System.Random()` and a random integer between 0 and a max value by using the method `Next`:

```
(* Given a max value, return a random non-neg int less than max*)
let random (max: int): int =
    let randomizer = System.Random() (* Get a new built-in Random
    object *)
    randomizer.Next(max) (* Return a random int between 0 and (max
    - 1) *)
```

Now, consider the version of `scramble` that builds upon three previous functions: `random`, `validRotation` and `rotate`.

```
(* Return a scrambled board rotated by m random legal rotations*)
let scramble (b: Board) (m: uint): Board =
    let (Board l) = b (* Deconstruct the Board, so l = char list
    *)
```

```

let rec r (count: int) (acc: Board) = (* Rotate a random area
count times*)
let next = random 1.Length (* Get a random number between
0 and the length of the board *)

match next with
| _ when count = (int m) -> acc (* If we have rotated m
times already, return the result *)
| next when not (validRotation b (Position (uint next)))
-> r count acc (* If the planned rotation is not valid
ignore the result and go again *)
| next -> r (count + 1) (rotate acc (Position (uint next))
) (* Call again with incremented counter and performed
rotation *)

r 0 b (* Call the function with an initial count of 0 and the
board *)

```

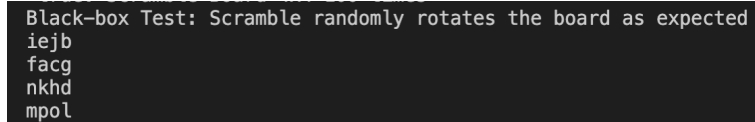
The `scramble` function successfully passed the tests. As it is based on the `random` function, I received different result each time. Consider the example of the print-statement below and in Figure 5. For the black-box and white-box tests of `scramble`, we refer to `blackboxtest.fsx` `whiteboxtest.fsx` respectively.

```

(* Scramble randomly rotates the board 10 times *)
let board = create 4u
let scrambled = scramble board 10u

printfn "%s" (board2Str scrambled)

```



```

Black-box Test: Scramble randomly rotates the board as expected
iejb
facg
nkhd
mpol

```

Figure 5: The result of the test for `scramble`

## 4 Final testing of Rotate

As already mentioned, the major hurdle of the task at hand was to implement the game logic or "game loop" that through the interaction with the player keeps running until the win-condition implemented in the `solved` function returns `true`. Such logic is implemented in the `gameloop` function below:

```

(* Gameloop keeps running in a loop while solved returns false *)
let rec gameloop (gameboard) =

```



```

if solved (gameboard) then
    printfn "Hooray"
    exit
else
    let input = System.Console.ReadLine(). (* Gets user input,
        returns a string *)
    let asUInt = uint input (* May throw an exception if it is
        not castable to number *)
    let asPosition = Position asUInt
    let gameboard = rotate gameboard asPosition
    printfn "%s" (board2Str gameboard)
    gameloop (gameboard)

```

gameloop (gameboard)

At the start of a new game, the created board should be scrambled, perhaps as much as 100 times to make sure it is challenging enough (cf. the file `game.fsx`):

```
let gameboard = (scramble (create 4u) 100u)
```

However, when testing the game by actually playing it, I set `scramble` to one random rotation of the board and have reached the `solved` configuration in just three successive steps at the rotation position 9 (cf. Figure 6):

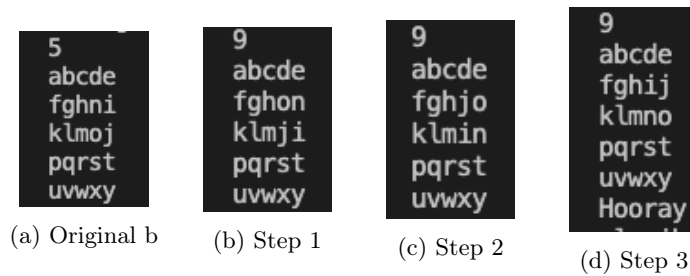


Figure 6: Playing Rotate in three steps

Voila! The application of the game `Rotate` was successful and the program works as expected. Yet, as it has been implied in task 7g5, there might be certain combinations of board-size  $n$  and maximum rotations  $m$  for which the computer (or a player) never steps out of the loop, presumably because the if-statement of `gameloop` is never being entered. In other words, that implies a scenario for a runtime error, and, in terms of theory, it should be approached as the infinite loop bug. Another possible scenario could also be related to the assumption that for some board combinations there might not be any feasible solutions. The discussion of the methods for solving those problems, however, goes beyond the scope of the present paper and will not be touched upon here.

## 5 Conclusion

The aim of the report at hand was to give an account of the considerations behind the design and implementation of the program for the puzzle-game `Rotate`. The puzzle should represent a square chess-like board of  $n \times n$ ,  $n \in \{2, 3, 4, 5\}$  fields, each having a unique position, and be associated with a unique letter of the English alphabet from 'a' to 'z'. Moreover, it should be able to take input from the player in the form of the board size and rotation position and, if valid, apply the corresponding rotation again and again until the puzzle has reached the `solved` configuration. The latter should correspond to the 25 letters from the English alphabet arranged alphabetically.

As far as the architecture behind the game is concerned, the program makes use of the two sum-types `Board` and `Position` defined in terms of a list of characters and an integer, respectively. The application of the `Rotate` also includes a number of predefined functions, responsible for different aspects of the game. For instance, the `validRotation` determines valid and invalid rotation positions.

Speaking about data structure, the program at hand makes use of lists and the functions `rotate`, `scramble` and `gameloop` make use of the recursion rather than loops.

Prior to writing the program, I had to run black-box tests on functions yet to be implemented. I approached that task by coding incrementally — i.e., constantly switching between writing bits of functions and printing the results to the console.

In terms of the implementation of `Rotate`, its game logic requires that the game keeps running in a loop by iteratively taking the position input from the player and applying the corresponding rotation, when valid, while the `solved` function returns `false`. The final testing of the game showed that it works as intended: (1) before a new game it scrambles the original board randomly the desired amount of times; (2) takes the board size input from the player and builds the board of the corresponding size; (3) takes the position input from the player and applies the corresponding rotation of letters, if the position is valid; and finally (4) returns "Hooray!" if the board has reached the `solved` configuration.

A final remark should be made on the issue of the application of `Rotate` in the future. The tests have been carried out on many — but by no means all — possible combinations of board-size  $n$  and maximum rotations  $m$ . That leaves us with a hypothetical "danger" of the game crashing for some game scenarios, which we believe should be addressed by the developers in the future.