PoP - Ugeopgave 7

Christoffer, Inge og Pernille

November 9, 2018

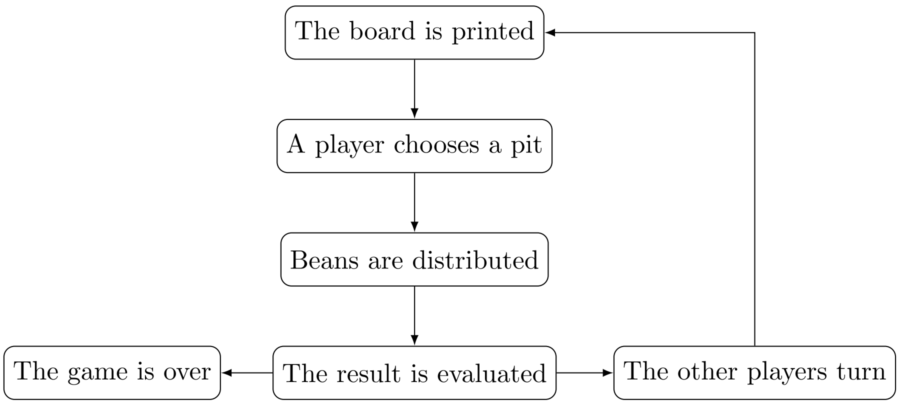

Figure 1: Flow of Awari

Preface

As a part of the Programming and Problem Solving course, we, three Computer Science students at Copenhagen University, build the game Awari.

Awari

Awari is an ancient two player game that resembles the beloved Kalaha. The main objective of Awari is to capture the most beans. The game consists of a board with 6 pits and one home pit for each player. Each of the 6 x 2 pits consists of 3 beans. The players must in turn take the amount of beans from a pit on his or her side of the board and distribute them in the following pits. The game continues until one of the two players has no beans left, and the winner is the player who captured the most beans in his or her homepit.

Problem description

We have implemented the Awari using the functional programming language F#. We were given two functions as a start, a turn and a play function, and a signiture file with type indication for several minor functions. To be able to play the game we have created functions to print the board, to distribute the beans, to check for game over etc. All the functions will be elaborated in this rapport.

Problem analysis and design

Game board and flow

We designed our programme starting by focusing on the game board. The board is the center of the game and thus the main part of our functions takes the board as input and returns a new board or evaluates the pits at the board. Figure 1 illustrates the simple flow of the game, where the players start by viewing the game board and then afterwards starts playing. As we see from Figure 1) the player returns to the printed board after every turn. Based on this, we started by creating our board. It consists of an int array with 14 integers whereas two are home pits and six pits for each player. As an array is mutable we preferred this type to lists. Moreover, the design of the game board was important to us. We wanted the gui design to be nice and simple. Therefore we started by focusing on the `printBoard` function.

It prints the array formed as a game board. It is more clear to the player that it is a game board an not just an array of integers. All

```
         | 3 | 3 | 3 | 3 | 3 | 3 |
|   0    |                       | 0 |
         | 3 | 3 | 3 | 3 | 3 | 3 |
```

Figure 2: Board of Awari

the pits are clearly divided and the home pits
are situated at each end of the board starting with zero beans.

The Players

The game is played by two players. We have created a type player consisting of a player1 and player2. It is important to distinguish between the two as they have each their home pit and each their side of the board, meaning that they have a specific index as their pits. Player1 has index 0-5 and has homepit in index 6. Player2 has index 7-12 and home pit in index 13. To solve this issue, we made a function `isHome` that was able to recognize which home pit belonged to each of the players.

We also needed to know the player's choice of pit from which he or she wanted to take beans. We decided that it should be possible for each player to enter a number on the keyboard between 1-6 and then the game had to convert it to the correct index number, instead of just letting the player know which index numbers belong to him or her. We created the function `getMove` to sove this problem. It matches the player and the input from a pressed keyboard and returns an index.

Playing the game

Another function we focused on was the `distribute` function as it distributes the beans taken from a pit and placed in the following. Thereby it updates the board and secure the flow in the game.

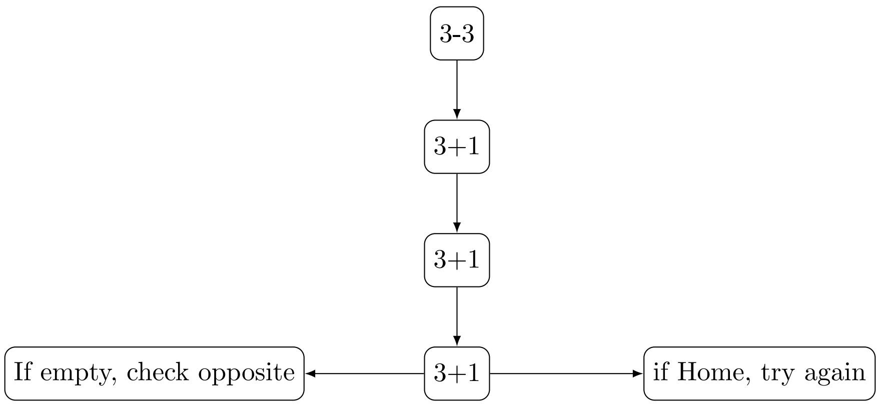

Figure 3: Distribute function

Appendix

```
module Awari

/// A game is played between two players
type player = Player1 | Player2

/// A board consisting of pits.
type board = int array

/// Each player has a set of regular pits and one home pit. A pit holds zero or
/// more beans
type pit = int

(*DOCUMENTATION OF printBoard*)
/// <summary>
/// Prints the board
/// </summary>
/// <param name="b"> A board to be printed </param>
/// <returns>() - it just prints</returns>
/// , e.g.,
/// <remarks>
/// Output is for example,
/// <code>
///
3
3
3   3   3   3
///
0
0
///
3
3
3   3   3   3
/// </code>
/// Where player 1 is bottom row and rightmost home
/// </remarks>
let printBoard (b: board) =
  System.Console.Clear ()
  let esc = string (char 0x1B)
  printf "
|"
  for i = 12 downto 7 do
      printf "%2i |" b.[i]
  printfn ""
  printf "| %2i |
| %i |\n" b.[13] b.[6]
  printf "
|"
  for i = 0 to 5 do
      printf "%2i |" b.[i]
  printfn ""


(*DOCUMENTATION OF isGameOver*)
/// <summary>
/// Checks whether the game is over
/// </summary>
/// <param name="b"> A board to check</param>
/// <returns>True if either side has no beans</returns>
let isGameOver (b: board) : bool =
  match b with
  | b when Array.forall (fun b -> (b = 0)) b.[0..5] -> true
  | b when Array.forall (fun b -> (b = 0)) b.[7..12] -> true
  | b -> false



(*DOCUMENTATION OF isHome*)
/// <summary>
/// Checks whether a pit is the player's home
/// </summary>
/// <param name="b">A board to check</param>
/// <param name="p">The player, whos home to check</param>
/// <param name="i">A regular or home pit of a player</param>
/// <returns>True if either side has no beans</returns>

let isHome (b: board) (p: player) (i: pit) : bool =
  match i with
  | 6 when p = Player1 -> true
  | 13 when p = Player2 -> true
  | _ -> false


(*DOCUMENTATION OF getMove*)
/// <summary>
/// Takes the pressed key as input and finds the pit of next move from the user.
/// </summary>
/// <param name="b">The board the player is choosing from</param>
/// <param name="p">The player, whose turn it is to choose</param>
/// <param name="q">The string to ask the player</param>
/// <returns>The indexnumber of the pit the player has chosen</returns>

let rec getMove (b:board) (p:player) (q:string) : pit =
  printfn "%s Choose a pit between 1-6" q
  let n = int (System.Console.ReadLine ())
  if (1 <= n && n <= 6) then
    match p with
    | Player1 when not (b.[n-1] = 0) -> n-1
    | Player2 when not (b.[n+6] = 0) -> n+6
    | _ -> printfn "This pit is empty. Try again."
           getMove b p q
  else
    printfn "This is not a valid input. Try again."
    getMove b p ""

(*DOCUMENTATION OF checkOpp*)
/// <summary>
/// Checks pit opposit of finalPit
/// </summary>
/// <param name="b"> A board to check</param>
/// <param name="i">The indexnumber of the finalPit of the player who just
/// played his/her turn</param>
/// <returns>The number of beans in the pit opposite of the finalPit</returns>

let checkOpp (b:board) (i: pit) : bool =
  if i = 13 then false
  elif i = 6 then false
  else
    let Opps = (b.Length - 2) - i
    (b.[Opps] <> 0)

(*DOCUMENTATION OF finalPitPlayer*)
/// <summary>
/// Checks whether Player1 or Player2 is the player of the final pit.
/// </summary>
/// <param name="i">The indexnumber of the finalPit of the player who just
/// played his/her turn</param>
/// <returns>Player1 or Player2</returns>

let finalPitPlayer (i: pit) : player =
  match i with
  | i when i <= 6 -> Player1
  | i -> Player2


(*DOCUMENTATION OF distribute*)
/// <summary>
/// Distributing beans counter clockwise, capturing when relevant
/// </summary>
/// <param name="b">The present status of the board</param>
/// <param name="p">The player, whos beans to distribute</param>
/// <param name="i">The regular pit to distribute</param>
/// <returns>A new board after the beans of pit i has been distributed, and which player
//val distribute : b:board -> p:player -> i:pit -> board * player * pit

let rec distribute (b:board) (p:player) (i:pit) : board * player * pit =
  let mutable j = i + 1
  ///Let k be the number of pits to distribute
  let mutable k = b.[i]
  while k > 0 do
    if (j <= 13) then
      b.[j] <- (b.[j] + 1)
      k <- k - 1
    if (j > 13) then
      j <- 0
    elif k = 0 then
      j <- j
    else
      j <- j + 1
  let finalPit = j
  if (checkOpp b finalPit) && (finalPitPlayer finalPit) = p && b.[finalPit] = 1 then
    let Opps = (b.Length - 2) - finalPit
    match p with
    | Player1 -> b.[6] <- b.[6] + b.[Opps] + b.[finalPit]
    | Player2 -> b.[13] <- b.[13] + b.[Opps] + b.[finalPit]
    b.[finalPit] <- 0
    b.[Opps] <- 0
  b.[i] <- 0
  (b, (finalPitPlayer finalPit), finalPit)

(*DOCUMENTATION OF turn*)
/// <summary>
/// Interact with the user through getMove to perform a possibly repeated turn of a play
/// </summary>
/// <param name="b">The present state of the board</param>
/// <param name="p">The player, whose turn it is</param>
/// <returns>A new board after the player's turn</returns>


let turn (b : board) (p : player) : board =
  let rec repeat (b: board) (p: player) (n: int) : board =
    printBoard b
    let str =
      if n = 0 then
        sprintf "%A's move. " p
      else
        "Again "
    let i = getMove b p str
    let (newB, finalPitsPlayer, finalPit)= distribute b p i
    if not (isHome b finalPitsPlayer finalPit)
       || (isGameOver b) then
      newB
    else
      repeat newB p (n + 1)
  repeat b p 0
```

```


(*DOCUMENTATION OF play*)
/// <summary>
/// Play game until one side is empty
/// </summary>
/// <param name="b">The initial board</param>
/// <param name="p">The player who starts</param>
/// <returns>A new board after one player has won</returns>


let rec play (b : board) (p : player) : board =
  if isGameOver b then
    let esc = string (char 0x1B)
    if b.[6] > b.[13] then
      System.Console.WriteLine(esc + "[31;1m" + "Game over. The winner is Player 1" + es
    elif b.[6] = b.[13] then
      System.Console.WriteLine(esc + "[33;1m" + "Game over. It's a tie" + esc + "[0m")
    else
      System.Console.WriteLine(esc + "[31;1m" + "Game over. The winner is Player 2" + es
    //printfn "Game over."
    b
  else
    let newB = turn b p
    let nextP =
      if p = Player1 then
        Player2
      else
        Player1
    play newB nextP
```