Progetto di programmazione funzionale e logica

Tommaso Cicco

Anno accademico 2021/2022

Indice

1	Spec	cifica del Problema	3
2	Ana 2.1	lisi del problema Analisi dei dati di input	4
	2.2	Analisi dei dati di output	4
	2.3	Relazioni intercorrenti tra i dati	4
3	Pros	gettazione dell'algoritmo	5
	3.1	Scelte di progetto	5
	3.2	Passi dell'algoritmo	5
4	Implementazione dell'algoritmo 7		
-	4.1	Implementazione in Haskell	7
	4.2	-	12
5	Test	ing dell'implementazione in Haskell	17
J	5.1	8 1	17
	5.2		17
	5.3		17
	5.4		18
	5.5		18
	5.6		19
	5.7		19
	5.8		19
	5.9	Inizio di nuova partita	20
	5.10		20
6	Test	ing dell'implementazione in Prolog	22
	6.1		22
	6.2		22
	6.3		22
	6.4	Validazione dell'input (casella occupata)	23
	6.5	Vittoria del giocatore X	23
	6.6	Vittoria del giocatore O	23
	6.7	00	24
	6.8		24
	6.9	1	25
	6.10	Uscita dal programma	25

1 Specifica del Problema

Il gioco del tris si svolge su una tabella 3×3 in cui si alternano le mosse di due giocatori indicati rispettivamente con X e O. Il gioco si conclude con la vittoria di X (risp. O) se sono stati messi tre simboli X (risp. O) sulla stessa riga, colonna o diagonale. Scrivere un programma che acquisisce da tastiera una mossa alla volta mostrando sullo schermo il contenuto aggiornato della tabella e determina la vittoria di uno dei due giocatori o il pareggio stampando sullo schermo un apposito messaggio.

2 Analisi del problema

2.1 Analisi dei dati di input

Ad ogni turno di gioco, i dati in ingresso sono rappresentati dalla scelta di una casella sulla tabella da parte di uno dei giocatori.

2.2 Analisi dei dati di output

Ad ogni turno di gioco, i dati in uscita sono rappresentati dallo stato della tabella, e, nel caso in cui si raggiunga una delle condizioni di fine partita, dal messaggio che comunica ai giocatori il vincitore o il pareggio.

2.3 Relazioni intercorrenti tra i dati

- Lo stato della tabella è aggiornato ad ogni turno secondo la scelta del giocatore. Il nuovo stato sarà uguale a quello precedente con il simbolo del giocatore presente nella casella scelta.
- La fine di una partita è determinata dal raggiungimento di una condizione di vittoria oppure di pareggio.
- La condizione di vittoria è raggiunta nel momento in cui sulla tabella sono presenti tre simboli identici sulla stessa riga, colonna o diagonale.
- La condizione di pareggio è raggiunta nel momento in cui non sono più presenti caselle libere e non è stata raggiunta una condizione di vittoria.

3 Progettazione dell'algoritmo

3.1 Scelte di progetto

- Per rappresentare la tabella di gioco si utilizzerà il tipo di dato lista, i cui elementi rappresenteranno le nove caselle a partire dalla prima riga.
- La lista sarà inizialmente costituita da elementi segnaposto, i quali rappresentano una casella libera; alla fine di ogni turno sarà generata una nuova lista con il segnaposto corrispondente alla casella scelta sostituito dal simbolo del giocatore cha ha appena effettuato una mossa.
- Per consentire la scelta di una casella, le righe della tabella saranno indicizzate da una lettera (A, B, C) e le colonne da un numero (1, 2, 3); una mossa valida sarà rappresentata dagli indici di riga e di colonna di una casella libera (esempio: A2).
- Al termine di ogni partita, sarà data la possibilità ai giocatori di iniziarne una nuova senza dover riavviare il programma.

3.2 Passi dell'algoritmo

- 1. Acquisire la mossa di un giocatore:
 - Controllare che la mossa inserita rappresenti una casella (A1 C3).
 - Controllare che la casella scelta dal giocatore sia libera.
 - Quando un controllo fallisce, l'acquisizione è ripetuta.
- 2. Aggiornare lo stato della tabella:
 - Generare una nuova lista sostituendo il segnaposto corrispondente alla casella scelta con il simbolo del giocatore corrente.
- 3. Controllare il raggiungimento di una condizione di vittoria:
 - Controllare ogni trio di elementi della lista che costituisce una riga, colonna, o diagonale sulla tabella.
 - Se è presente un trio di elementi identici (diversi dal segnaposto), il giocatore ha vinto.
- 4. Se non è stata raggiunta una condizione di vittoria, controllare il raggiungimento di una condizione di pareggio:
 - Controllare che la lista non contenga elementi segnaposto.
- 5. Visualizzare la tabella aggiornata.
- 6. Se non è stata raggiunta una condizione di fine partita, acquisire la mossa del giocatore successivo (punto 1).

- 7. Se è stata raggiunta una condizione di fine partita, visualizzare il messaggio adatto.
- 8. Chiedere all'utente se vuole giocare di nuovo:
 - Se la risposta è affermativa, rigenerare la lista iniziale e continuare l'esecuzione dal punto 1.
 - $\bullet\,$ In caso contrario, terminare l'esecuzione.

4 Implementazione dell'algoritmo

4.1 Implementazione in Haskell

```
import Data.List (intercalate)
import Data.Char (toLower)
{- Definizioni di nuovi tipi -}
{- Box rappresenta una casella della tabella: puo' essere vuota o
    occupata da un giocatore (X o O).
Una casella vuota e' rappresentata dal placeholder (-) -}
data Box = Empty | Full Player
   deriving Eq
{- Player rappresenta il simbolo di un giocatore -}
data Player = X | 0
   deriving (Show, Eq)
{- Per visualizzare le mosse sulla tabella creiamo un'istanza della
    typeclass Show per il tipo Box
gli spazi ai lati sono aggiunti per centrare i simboli nelle rispettive
    caselle -}
instance Show Box where
   show Empty = " - "
   show (Full X) = " X "
   show (Full 0) = " 0 "
{- Definizioni di funzioni-}
{- Tabella vuota utilizzata all'inizio di una partita -}
emptyBoard :: [Box]
emptyBoard = [Empty, Empty, Empty,
            Empty, Empty, Empty,
            Empty, Empty, Empty]
{- Ritorna una riga della tabella sotto forma di stringa: le caselle
    sono separate dal carattere "|" -}
renderRow :: [Box] -> String
renderRow row = intercalate "|" $ map show row
```

```
{- Stringa separatrice tra righe -}
dividingLine :: String
dividingLine = " ____|__\n" ++ " | "
{-Indici di colonna -}
topCoordsLine :: String
topCoordsLine = " 1
                             3\n"
{- Funzione per visualizzare la tabella -}
renderBoard :: [Box] -> IO ()
renderBoard board = do
 putStrLn topCoordsLine
 putStrLn $ "A " ++ renderRow firstRow
 putStrLn dividingLine
 putStrLn $ "B " ++ renderRow secondRow
 putStrLn dividingLine
 putStrLn $ "C " ++ renderRow thirdRow
 where firstRow = take 3 board
       secondRow = drop 3 . take 6 $ board
       thirdRow = drop 6 board
{- Funzione che mappa una mossa valida al corrispondente indice di
    casella, restituisce -1 per mosse invalide -}
getBoxIndex :: String -> Int
getBoxIndex "a1" = 0
getBoxIndex "a2" = 1
getBoxIndex "a3" = 2
getBoxIndex "b1" = 3
getBoxIndex "b2" = 4
getBoxIndex "b3" = 5
getBoxIndex "c1" = 6
getBoxIndex "c2" = 7
getBoxIndex "c3" = 8
getBoxIndex _ = -1
{- Funzione che controlla se una casella e' occupata -}
isEmpty :: Box -> Bool
isEmpty Empty = True
isEmpty (Full _) = False
```

```
{- Funzione che data una tabella e un indice, controlla che la casella a
    quell'indice sia libera, in caso contrario restituisce -1 -}
validatePosition :: [Box] -> String -> Int
validatePosition board m | i /= -1 && isEmpty (board !! i) = i
                      | otherwise = -1
   where i = getBoxIndex m
{- Funzione che prende una lista e un indice, e restituisce una coppia
    formata dalle liste degli elementi a sinistra e a destra di quello
    all'indice specificato -}
splitAtIndex :: [a] -> Int -> ([a], [a])
splitAtIndex lst i = (left, right)
   where
       (left, xs) = splitAt i lst
       right = drop 1 xs
{- Funzione per l'inserimento di un simbolo Player nella tabella di
    gioco -}
playMove :: [Box] -> Box -> Int -> [Box]
playMove board player i = xs ++ [player] ++ ys
   where (xs, ys) = splitAtIndex board i
{- Funzione che prende una mossa dall'utente e restituisce l'indice di
    tabella corrispondente -}
getMove :: [Box] -> Player -> IO Int
getMove board currentPlayer = do
   putStrLn $ "\nPlayer " ++ show currentPlayer ++ " enter a move (A1 -
       C3): "
   move <- getLine
   let index = validatePosition board $ map toLower move
   if index /= -1
       then return index
   else do
       putStrLn "\nInvalid move, try again!"
       getMove board currentPlayer
{- Funzione che gestisce l'alternanza dei turni -}
nextPlayer :: Player -> Player
```

```
nextPlayer X = 0
nextPlayer 0 = X
{- Funzione che controlla se nella tabella attuale e' presente una
    combinazione vincente -}
checkWinner :: Player -> [Box] -> Bool
checkWinner move board =
   or [head board == Full move && board !! 1 == Full move && board !! 2
       == Full move,
       board !! 3 == Full move && board !! <math>4 == Full move && board !! 5
           == Full move,
       board !! 6 == Full move && board !! 7 == Full move && board !! 8
           == Full move,
       head board == Full move && board !! 3 == Full move && board !! 6
           == Full move,
       board !! 1 == Full move && board !! 4 == Full move && board !! 7
           == Full move,
       board !! 2 == Full move && board !! 5 == Full move && board !! 8
           == Full move,
       head board == Full move && board !! 4 == Full move && board !! 8
           == Full move.
       board !! 6 == Full move && board !! 4 == Full move && board !! 2
           == Full move]
{- Funzione che controlla se la partita e' un pareggio (tutte le caselle
    occupate da un giocatore e nessuna condizione di vittoria) -}
checkTie :: [Box] -> Bool
checkTie board = all (\box -> not (isEmpty box)) board
{- Funzione che comunica all'utente l'evento di un pareggio o di una
    vittoria, in caso contrario chiama ricorsivamente gameLoop con il
    giocatore successivo -}
checkGameState :: [Box] -> Player -> IO ()
checkGameState board currentPlayer | checkWinner X board = putStrLn
    "\nPlayer X won!"
                                | checkWinner O board = putStrLn
                                    "\nPlayer 0 won!"
                                | checkTie board
                                                  = putStrLn "\nIt's a
                                    tie!"
                                | otherwise = gameLoop board (nextPlayer
                                    currentPlayer)
{- Funzione principale che gestice lo stato della partita-}
```

```
gameLoop :: [Box] -> Player -> IO ()
gameLoop board currentPlayer = do
   choice <- getMove board currentPlayer</pre>
   {- Creiamo una nuova tabella con il simbolo inserito dal giocatore -}
   let newBoard = playMove board (Full currentPlayer) choice
   renderBoard newBoard
   {- Controlliamo un'eventuale vittoria o pareggio, se nessuno di
        questi eventi si verifica sara' richiamato ricorsivamente
        gameLoop con il giocatore successivo -}
   checkGameState newBoard currentPlayer
{- Funzione che chiede all'utente se vuole fare un'altra partita, se
    l'utente accetta e' richiamata main -}
newGame :: IO ()
newGame = do
   putStrLn "\nPlay again? (y/n):"
   c <- getChar
   let choice = toLower c
   if c == 'y' then main
   else if c == 'n' then do
       putStrLn "\nBye Bye!"
   else do
       putStrLn "\nInvalid character, try again!"
       {\tt newGame}
{- Funzione main, da eseguire per lanciare il programma -}
main :: IO ()
main = do
   putStrLn "\nWelcome to tic tac toe, good luck and have fun!\n"
   renderBoard emptyBoard
   {\tt gameLoop\ emptyBoard\ X}
   {- Alla fine di una partita chiediamo all'utente se vuole giocare di
        nuovo -}
   newGame
```

4.2 Implementazione in Prolog

```
/* Predicati per rappresentare lo stato delle caselle */
\% Una casella e' occupata da un giocatore se contiene una x oppure una o
x(B) : - B = x.
o(B) : - B = o.
player(B) :- x(B); o(B).
\% Una casella e' piena se e' occupata da un giocatore, in caso contrario
    e' vuota
full(B) :- player(B).
empty(B) :- +(full(B)).
/* Predicati per la visualizzazione dello stato della tabella */
% Visualizzazione dello stato delle caselle
showBox(B) :- o(B), write(' 0 ').
showBox(B) :- x(B), write(' X ').
showBox(B) :- empty(B), write(' - ').
% Visualizzazione dello stato delle righe
showRow(X, Y, Z) :-
   showBox(X),
   write('|'),
   showBox(Y),
   write('|'),
   showBox(Z), nl.
% Visualizzazione della riga separatrice
showDividingLine :-
   write(' ____|___'), nl,
write(' | '), nl.
% Visualizzazione degli indici di colonna
showColIndex :-
   write(' 1
                    2
                         3'), nl, nl.
% Visualizzazione dello stato della tabella corrente
showBoard([A, B, C, D, E, F, G, H, I]) :-
```

```
showColIndex,
   write('A'),
   showRow(A, B, C),
   showDividingLine,
   write('B'),
   showRow(D, E, F),
   showDividingLine,
   write('C'),
   showRow(G, H, I).
/* Validazioni dell'input */
% Validazione di una mossa inserita dall'utente (a1 - c3)
isValid(M) :-
   M == 'a1';
   M == 'a2';
   M == 'a3';
   M == 'b1';
   M == b2;
   M == 'b3';
   M == 'c1';
   M == 'c2';
   M == 'c3'.
% Controlla se la casella all'indice specificato e' vuota
% I: indice di lista (1 - 9)
\% B: lista che rappresenta la tabella di gioco corrente
isEmpty(I, B) :-
   nth(I, B, E), % Successo se l'I-esimo elemento di B e' uguale ad E
   empty(E).
% Predicato per ottenere l'indice corrispondente ad una mossa
getBoxIndex(M, I) :-
   (M == 'a1', I is 1);
   (M == 'a2', I is 2);
   (M == 'a3', I is 3);
   (M == 'b1', I is 4);
   (M == b2', I is 5);
   (M == 'b3', I is 6);
   (M == 'c1', I is 7);
   (M == c2', I is 8);
   (M == c3', I is 9).
/* Logica di gioco */
% Sostituzione dell'N-esimo elemento di una lista con un nuovo elemento
```

```
% Argomenti: indice, elemento da inserire, lista corrente, nuova lista
% Caso base: sostitizione del primo elemento
\% Caso ricorsivo: se l'indice e' maggiore di 1, si richiama il predicato
    sulle code delle liste con I diminuito di 1, fino a ricondursi al
    caso base
replaceAtIndex(1, E, [_ | T], [E | T]).
replaceAtIndex(I, E, [H | T], [H | R]) :-
   I > 1,
   INew is I - 1,
   replaceAtIndex(INew, E, T, R).
% Predicato per ottenere una mossa dall'utente e conoscere l'indice
    corrispondente
% B: lista che rappresenta la tabella di gioco corrente
% P: giocatore attivo nel turno corrente (x oppure o)
getMove(B, P, I) :-
   player(P),
   ((x(P),
                    % Turno del giocatore X
   nl, write('Player X enter a move (a1 - c3): '));
            % Turno del giocatore O
   nl, write('Player 0 enter a move (a1 - c3): '))),
   read(M),
   isValid(M),
                    % Controlla che la mossa inserita sia valida
   getBoxIndex(M, I1), % Ottiene il corrispondente indice di lista
   I is I1.
getMove(B, P, I) :-
   player(P),
   nl, write('Invalid move, try again!'),nl,
   getMove(B, P, I).
% Predicato per gestire l'alternanza dei turni
nextPlayer(x, o).
nextPlayer(o, x).
% Controlla se e' stata raggiunta una condizione di vittoria
checkWin(B, P) :-
   rowWin(B, P);
   colWin(B, P);
   diagWin(B, P).
% Condizioni di vittoria
% 3 simboli uguali allineati in riga
rowWin(B, P) :-
   player(P),
```

```
(B = [P, P, P, _, _, _, _, _, _];
    B = [_, _, _, _, P, P, P, _, _, _];
    B = [_, _, _, _, _, _, _, P, P]).
\% 3 simboli uguali allineati in colonna
colWin(B, P) :-
   player(P),
   (B = [P, \_, \_, P, \_, \_, P, \_, \_];
    B = [_, P, _, _, P, _, _, P, _];
    B = [_, _, P, _, _, P, _, _, P]).
\% 3 simoli uguali allineati in diagonale
diagWin(B, P) :-
   player(P),
   (B = [P, _, _, P, _, _, P];
    B = [_, P, _, _, P, _, P, _];
    B = [\_, \_, P, \_, P, \_, P, \_, \_]).
% Controlla se e' stata raggiunta la condizione di parita' (tutte le
    caselle occupate da un giocatore)
checkTie(B) :- maplist(player, B).
\% Predicato che controlla lo stato del gioco ad ogni turno comunicando
    all'utente un'eventuale vittoria o pareggio
% Se non e' riscontrata una condizione di fine partita, viene chiamato
    ricorsivamente gameLoop per iniziare un nuovo turno
% Controlla vittoria del giocatore x
checkGameState(B, P) :-
   checkWin(B, P),
   x(P),
   nl, write('Player X won!').
% Controlla vittoria del giocatore o
checkGameState(B, P) :-
   checkWin(B, P),
   o(P),
   nl, write('Player 0 won!').
% Controlla parita'
checkGameState(B, _) :-
   checkTie(B),
   nl, write('It\'s a tie!').
% Se tutte le precedenti falliscono, continua con il prossimo turno
checkGameState(B, P) :-
   nextPlayer(P, NP),
   gameLoop(B, NP).
% Predicato principale per la gestione della partita
gameLoop(B, P) :-
```

```
getMove(B, P, I),
                          % Acquisice una mossa da giocatore corrente
   replaceAtIndex(I, P, B, NB), % Genera una nuova lista con la mossa
       inserita dal giocatore
   showBoard(NB),
                            % Sostra la nuova tabella
   condizione di fine partita
% Predicato per richedere all'utente se vuole giocare ancora una volta
   finita una partita
\% Viene invocato una volta raggiunta una condizione di fine partita
    (parita' oppure vittoria)
newGame :-
   nl, write('Play again? (y/n): '),
   read(C),
   ((C == 'y', % Se il carattere letto e' y, chiama ricorsivamente la
       funzione main
    main);
   (C == 'n', % Altrimenti, se il carattere letto e' n, termina
    nl, write('Bye, Bye!'))).
newGame :-
   nl, write('Invalid character, try again!'),
   newGame.
% Predicato main, da eseguire per lanciare il programma
   nl, write('Welcome to tic tac toe, good luck and have fun!'), nl,
   showBoard([e,e,e,e,e,e,e,e,e]), % Stampa la tabella vuota
   gameLoop([e,e,e,e,e,e,e,e,e], x), % Inizia il gioco con la tabella
       {\tt vuota} e il giocatore {\tt x}
   newGame.
```

5 Testing dell'implementazione in Haskell

5.1 Acquisizione corretta (mossa del giocatore X)

Player O enter a move (A1 - C3):

5.2 Acquisizione corretta (mossa del giocatore O)

Player O enter a move (A1 - C3):

c1

1 2 3

A - | - | - |

----|
| B - | X | | ----|
| C O | - | -

Player X enter a move (A1 - C3):

5.3 Validazione dell'input (mossa inesistente)

Player X enter a move (A1 - C3): A-1

Invalid move, try again!

Player X enter a move (A1 - C3): 2,345 Invalid move, try again!

Player X enter a move (A1 - C3):

5.4 Validazione dell'input (casella occupata)

Player X enter a move (A1 - C3): a1

1 2

A X | - | -

Player O enter a move (A1 - C3):

Invalid move, try again!

Player O enter a move (A1 - C3):

5.5 Vittoria del giocatore X

Player X enter a move (A1 - C3):

a3

1 2 3



Player X won!

Play again? (y/n):

5.6 Vittoria del giocatore O

Player O enter a move (A1 - C3): c1

1 2

A X | - | 0

B X | O | -

C O | X | -

Player 0 won!

Play again? (y/n):

5.7 Pareggio

Player X enter a move (A1 - C3): c3

1 2 3

A X | O | X

B X | O | O

 $C \quad O \quad X \quad X$

It's a tie!

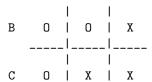
Play again? (y/n):

5.8 Vittoria con l'ultima casella

Player X enter a move (A1 - C3): c3

1 2 3

A X | O | X



Player X won!

Play again? (y/n):

5.9 Inizio di nuova partita

Player X won!

Play again? (y/n):

У

Welcome to tic tac toe, good luck and have fun!

1 2 3

A - | - | -____|___

B - | - | -

C - | - | -

Player X enter a move (A1 - C3):

5.10 Uscita dal programma

Player X enter a move (A1 - C3):

c1

1 2

A X | O | X

B O | X | O

C X | - | -

```
Player X won!
Play again? (y/n):
n
Bye Bye!
*Main>
```

6 Testing dell'implementazione in Prolog

6.1 Acquisizione corretta (mossa del giocatore X)

Player X enter a move (a1 - c3): b2. 1 2 3

1 - | - | -

____|___|

C - | - | -

Player O enter a move (a1 - c3):

6.2 Acquisizione corretta (mossa del giocatore O)

Player O enter a move (a1 - c3): a1.

1 2

A 0 | - | -

B - | X | -

C - | - | -

Player X enter a move (a1 - c3):

6.3 Validazione dell'input (mossa inesistente)

Player X enter a move (a1 - c3): a32.

Invalid move, try again!

Player X enter a move (a1 - c3): 23451.

Invalid move, try again!

Player X enter a move (a1 - c3): ciao.

Invalid move, try again!

Player X enter a move (a1 - c3):

6.4 Validazione dell'input (casella occupata)

Player X enter a move (a1 - c3): a2

1 2

A O | X | -

B - | X | -

C - | | | - | -

Player O enter a move (a1 - c3): a2.

Invalid move, try again!

Player O enter a move (a1 - c3):

6.5 Vittoria del giocatore X

Player X enter a move (a1 - c3): c1.

1 2

A X | O | X

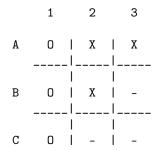
B 0 | X | -

X | - | O

Player X won!
Play again? (y/n):

6.6 Vittoria del giocatore O

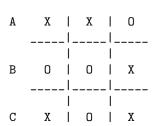
Player O enter a move (a1 - c3): c1.



Player 0 won!
Play again? (y/n):

6.7 Pareggio

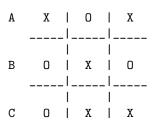
Player X enter a move (a1 - c3): c3. 1 2 3



It's a tie!
Play again? (y/n):

6.8 Vittoria con l'ultima casella

Player X enter a move (a1 - c3): c3. 1 2 3



Player X won!
Play again? (y/n):

6.9 Inizio di nuova partita

Player X won!
Play again? (y/n): y.

Welcome to tic tac toe, good luck and have fun!

1 2

B - | - | -| ----|

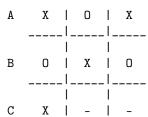
C - | - | -

Player X enter a move (a1 - c3):

6.10 Uscita dal programma

Player X enter a move (a1 - c3): c1.

1 2 3



Player X won!

Play again? (y/n): n.

Bye, Bye!

true ?

(1 ms) yes

| ?-