Relazione di Progetto d'Esame

Tommaso Petrelli

Università degli Studi di Urbino Carlo Bo
Insegnamento di Programmazione Logica e Funzionale
del Corso di Informatica Applicata

Sessione Estiva 2022/2023

Codice matricola: 305558
Anno di corso: Terzo

Indice

1 Specifica del Problema

Scrivere un programma Haskell e un programma Prolog che implementino il gioco del Tic-Tac-Toe interattivo a singolo giocatore, e che utilizzino l'algoritmo di Minimax [1] per far prendere decisioni ottime all'avversario (a cui si farà riferimento tramite il termine CPU).

[1] Si decide di non trattarne l'analisi considerandolo algoritmo noto. Verrà tuttavia descritto accuratamente nella sezione 3.1.

2 Analisi del Problema

2.1 Dati di Ingresso del Problema

I dati di ingresso del problema sono rappresentati da due insiemi finiti di mosse di lunghezza massima (risp. minima) pari a cinque (risp. tre), se l'insieme rappresenta le mosse fatte dal giocatore che inizia per primo, o a quattro (risp. due), nel caso opposto. Una delle regole del Tic-Tac-Toe asserisce che un giocatore non possa ripetere una mossa già precedentemente compiuta, motivo per il quale gli insiemi che rappresentano i dati di ingresso non possono contenere elementi doppi.

Ad esempio, si potrebbero avere le seguenti istanze di dati di ingresso (rappresentando le mosse come numeri interi):

- Insieme delle mosse del giocatore uno (inizia per primo):

$$G1 := \{1, 3, 8, 4, 9\}$$

- Insieme delle mosse del giocatore due:

$$G2 := \{5, 2, 6, 7\}$$

L'ordine degli elementi all'interno di questi insiemi è importante, per cui $G1' := \{9, 8, 3, 4, 1\}$ si considera come una diversa istanza di dati di ingresso rispetto a $G1$.

2.2 Dati di Uscita del Problema

Per il problema proposto si prevede un unico dato d'uscita, vale a dire, il risultato di una partita di Tic-Tac-Toe descritto da un insieme finito di massimo nove e minimo cinque elementi rappresentante la configurazione finale della griglia di gioco. Il risultato della partita potrà essere uno tra: vittoria, sconfitta e parità.

Ad esempio, considerando gli insiemi definiti nella sezione precedente, il risultato che otterremmo potrebbe essere descritto dalla seguente configurazione finale della griglia:

$$G := \{1, 2, 1, 1, 2, 2, 2, 1, 1\}$$

dove i termini 1 e 2 mostrano come la cella sia stata occupata da una mossa rispettivamente del primo o del secondo giocatore. Da G possiamo dedurre che il risultato della partita rappresenti una situazione di pareggio. Si sottolinea che, anche in questo caso, l'ordine degli elementi di G è fondamentale per decidere il risultato corretto.

2.3 Relazioni Intercorrenti tra i Dati del Problema

Le relazioni sfruttate per legare i dati del problema sono strettamente correlate alle regole del gioco Tic-Tac-Toe.

In primo luogo, è importante sottolineare come, tra i due insiemi rappresentanti i dati di

ingresso, vi deve essere intersezione vuota. Difatti, una tra le regole del gioco asserisce che è impossibile fare una mossa su una cella già occupata. Di conseguenza, le mosse non possono essere ripetute all'interno della stessa istanza del problema.

In secondo luogo, è altrettanto importante definire le relazioni tra gli elementi dell'insieme rappresentante la configurazione finale della griglia di gioco. Tali relazioni, infatti, permetteranno di generare una codifica capace di trasformare la configurazione della griglia nel risultato della partita. Anche in questo caso deriviamo tale relazione dalle regole del Tic-Tac-Toe.

Per esempio, si potrebbe immaginare di riempire l'insieme rappresentante la griglia di gioco G con tutti elementi neutri (e.g. 0) rendendo fissa a nove la sua dimensione. Così facendo, si possono ora numerare i suoi elementi, $s_i \in \{0, 1, 2\}$, da 1 a 9 (i.e. $G := \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$) ed esprimere le seguenti relazioni:

- se una tra le seguenti triple

$$(s_1, s_2, s_3),\ (s_4, s_5, s_6),\ (s_7, s_8, s_9),$$

$$(s_1, s_4, s_7),\ (s_2, s_5, s_8),\ (s_3, s_6, s_9),$$

$$(s_1, s_5, s_9),\ (s_3, s_5, s_7)$$

  contiene elementi identici (diversi da 0) allora la griglia G descrive una situazione di vittoria (e quindi anche di sconfitta);

- altrimenti, la griglia rappresenta una situazione di parità.

Come esempio, si vuole riprendere la configurazione finale G della sezione precedente per mostrare come possa essere tradotta in "parità". Data la numerazione appena descritta e l'insieme G, possiamo definire le seguenti triple

$$(1, 2, 1),\ (1, 2, 2),\ (2, 1, 1),$$

$$(1, 1, 2),\ (2, 2, 1),\ (1, 2, 1),$$

$$(1, 2, 2),\ (1, 2, 1)$$

da cui si può notare come in nessuna di esse si presenti una situazione analoga a $(1, 1, 1)$ o $(2, 2, 2)$. Si può concludere pertanto che G indichi un pareggio.

3 Progettazione dell'Algoritmo

3.1 Scelte di Progetto

I dati di ingresso sono stati descritti come insiemi finiti di mosse. Tali mosse saranno decise dai giocatori partecipanti ad una partita di Tic-Tac-Toe. Ovviamente, non si può pensare di chiedere ad un giocatore immediatamente la lista delle mosse che vorrebbe fare. Le decisioni di un giocatore, infatti, possono variare a seconda delle mosse fatte dal suo avversario. Per descrivere l'algoritmo che guiderà l'esecuzione del gioco si ha bisogno di sapere che cos'è una mossa, come si compone la griglia di gioco e come vengono acquisite le mosse dei giocatori.

Nel capitolo due si è proposto un insieme di elementi a scopo illustrativo come definizione di griglia di gioco. Difatti, si considera vantaggioso rappresentare la griglia di gioco mediante una struttura dati lineare come una lista; si decide inoltre di rendere fissa la dimensione di quest'ultima. Di conseguenza, si può affermare che tra le condizioni iniziali dell'algoritmo che descrive una situazione di gioco si ha bisogno di istanziare una lista finita composta da nove elementi, in cui ogni elemento rappresenta una cella e quindi una potenziale posizione in cui fare una mossa. Vale la pena dare una definizione a ciò che si intende quando si parla di mossa. Data la struttura della griglia di gioco, si può pensare che un ottimo modo di rappresentare una mossa sia quello di utilizzare numeri naturali presi dall'insieme $\{1, 2, 3, 4, 5, 6, 7, 8, 9\} \subset \mathbb{N}$. Di questa scelta si vogliono evidenziare specialmente due vantaggi:

- dato che la specifica del problema prevede che il gioco sia interattivo, rappresentare una mossa con un numero naturale appare essere molto intuitivo per un giocatore;
- risulta più semplice gestire una mossa utilizzando numeri naturali piuttosto che tipi di dato composti composti.

Il metodo scelto per rappresentare una mossa è anche completo, in quanto non ci interessa sapere altro senonché la posizione che la mossa andrà ad occupare nella griglia di gioco. In realtà un altro aspetto fondamentale è quello di conoscere il proprietario di una mossa. Questa informazione si considera tuttavia non concettualmente correlata da quella che invece è la rappresentazione di una mossa. In via del tutto convenzionale, si decide a priori di assegnare al giocatore pilotato dall'utente il simbolo '$\circ$', mentre le mosse fatte dall'avversario (chiamato CPU) saranno indicate attraverso il simbolo '$\times$'. Si può quindi dedurre che le celle della griglia di gioco andranno ad ospitare dei caratteri, potendo descrivere dunque la griglia di gioco come lista di caratteri di lunghezza nove. Infine, come carattere neutro che rappresenta una cella libera possiamo scegliere il carattere spazio (' ').

Seguendo le scelte di progetto appena descritte, è possibile passare alla descrizione della fase di acquisizione delle mosse dei giocatori. Per il giocatore pilotato dall'utente si ha bisogno di acquisire un numero intero e poi aggiornare opportunamente la lista che rappresenta la griglia di gioco. Ad esempio, se l'utente scegliesse la mossa '8', basterebbe aggiornare l'ottava cella della lista e proseguire con la partita. In questo caso si può notare come non si ha bisogno di altre strutture dati poiché andrebbero ad appesantire la memoria senza portare vantaggi computazionali e come la lista delle mosse fatte dal giocatore sia intrinsecamente contenuta nella lista che rappresenta la griglia.

Per il giocatore pilotato da CPU, al contrario, è necessario implementare un algoritmo che decida il processo di scelta di una mossa. L'algoritmo che si è scelto di utilizzare è quello che si basa sul metodo di decisione detto Minimax[2]. L'efficacia di questo algoritmo è data dal fatto che nel gioco del Tic-Tac-Toe si conoscono a priori quali siano i possibili risultati (vittoria, sconfitta, parità) così come in ogni momento si conoscono tutte le informazioni necessarie per raggiungerli. Inoltre, il Tic-Tac-Toe lo si classifica come un gioco a somma zero[3], caratterizzato dall'interazione strategica tra esattamente due giocatori. Si può definire l'algoritmo di Minimax come un algoritmo ricorsivo basato sul *backtracking* rispetto all'albero delle simulazioni di gioco. Assumendo che l'avversione faccia solo mosse ottime, lo scopo dell'algoritmo è quello di trovare la mossa migliore che un giocatore possa fare in una certa situazione di gioco. Per descrivere questo algoritmo è necessario definire due giocatori comunemente chiamati in letteratura MAX (o massimizzatore) e MIN (o minimizzatore). L'obbiettivo del giocatore MAX sarà quello di massimizzare il guadagno prodotto da una mossa, mentre quello di MIN è esattamente l'opposto. La generazione dell'albero delle simulazioni viene fatta in profondità e si origina dal nodo radice, a partire dal quale MAX farà la prima mossa; lo scopo è dunque quello di far arrivale al nodo radice il valore maggiore possibile. La mossa che produrrà tale valore sarà quella ottima e quindi quella verrà scelta da CPU. In generale, ogni mossa fatta produrrà un nuovo nodo figlio, vale a dire un nuovo stato della griglia di gioco, e ad ogni nodo sarà associato un valore. Una volta raggiunte le foglie dell'albero delle simulazioni sarà possibile iniziare a valutare i nodi a ritroso. Dato un nodo foglia, gli si assegna un valore di guadagno (e.g. 10), di perdita (e.g. -10), oppure di parità (e.g. 0) andando a guardare lo stato della griglia, la quale si troverà in una configurazione terminale. Dalle foglie si procede progressivamente a ritroso fino al nodo radice. Ad ogni punto di ramificazione si considerano i valori di tutti i nodi figli: se nel nodo padre il turno appartiene a MAX allora si lascia propagare il valore massimo trovato; altrimenti, si prende il valore minimo.
La versione dell'algoritmo vista finora potrebbe far scegliere a CPU una mossa che non è tuttavia quella ottima. CPU potrebbe infatti scegliere di fare una mossa che lo porterebbe ad una vittoria più lenta o ad una sconfitta più rapida. Per risolvere questo problema, viene assegnato il valore ad un nodo foglia, considerando anche la profondità a cui questo si trova all'interno dell'albero. Aggiungere la profondità nell'assegnazione del valore ad una foglia, aiuta infatti l'algoritmo a scegliere la mossa più ottimale. Questo si traduce nel sottrarre (risp. sommare) il livello di profondità nel caso in cui la foglia presenti una situazione di vittoria (risp. sconfitta) per CPU.

Per la progettazione dell'algoritmo di Minimax si prevede l'utilizzo di liste di lunghezza prefissata pari a nove per memorizzare tutti i valori dei nodi figli rispetto ad un nodo padre. Tali liste rappresenteranno certi stati della griglia di gioco, non contenendo però i caratteri '$\circ$', '$\times$' e *spazio*, ma i valori di guadagno/perdita. Se non si riesce a compiere una mossa poiché la cella corrispondente risulta occupata, si genera comunque un nodo foglia e si adotta la seguente convenzione: se nel nodo padre il turno appartiene al giocatore MAX, allora il valore assegnato al nodo foglia sarà di massima perdita (e.g. -1000); se nel nodo padre il turno appartiene al giocatore MIN verrà assegnato un valore di massimo guadagno (e.g. 1000). Questo meccanismo garantisce l'automatica esclusione di tali mosse illegali dall'insieme delle possibili mosse ottime.

[2]Minimax: https://it.wikipedia.org/wiki/Minimax
[3]Gioco a somma zero: https://it.wikipedia.org/wiki/Gioco_a_somma_zero

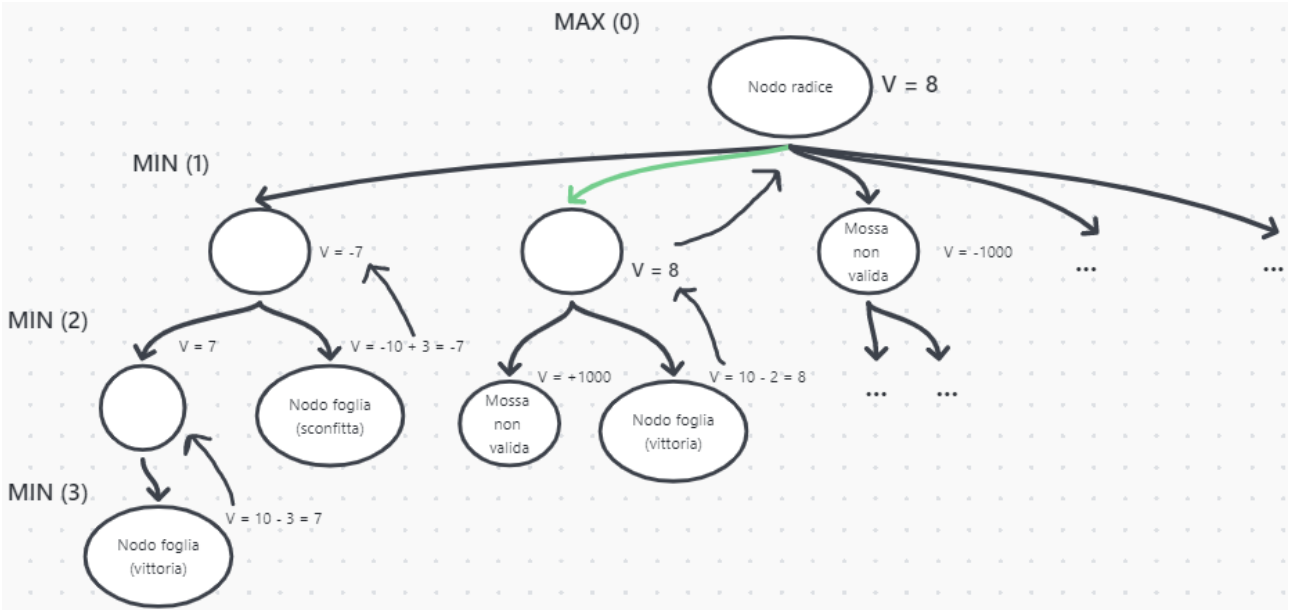

Figura 1: Esempio di albero delle simulazioni generato con Minimax in cui la mossa ottima calcolata è quella che genera un guadagno massimo pari a 8.

Si decide di non utilizzare sempre l'algoritmo di Minimax. Infatti, solamente per scegliere la prima mossa di CPU si farà riferimento alle strategie note del Tic-Tac-Toe. Quindi, si è deciso che CPU dovrà scegliere il prima possibile la cella centrale, altrimenti posizionare la sua prima mossa in una qualsiasi delle celle agli angoli della griglia (e.g. quella in alto a sinistra indicata dal indice 1).

Infine, si prevede una validazione sull'input dell'utente nel momento in cui sceglie la sua mossa. Se il numero rappresentante la mossa non dovesse essere compreso tra 1 e 9, non verrà infatti accettato. Inoltre, la mossa non sarà accettata nemmeno nel caso in cui venga specificato un numero corrispondente ad una cella già occupata. La mossa di CPU non verrà sottoposta a controlli in quanto la sua validità sarà garantita dalla correttezza dell'algoritmo che la decide.

3.2 Passi dell'Algoritmo

I passi dell'algoritmo per risolvere il problema sono i seguenti:

- Inizializzare vuota la griglia di gioco.
- Finché la partita non termina:
  - Verificare che l'avversario non abbia vinto.
  - Acquisire la mossa del giocatore.
    - *Se il giocatore è l'utente*: acquisire la mossa verificando che sia valida.
    - *Se il giocatore è CPU*: calcolare la mossa ottima.
      - Prima mossa: scegliere cella centrale, o, se occupata, scegliere uno qualsiasi degli angoli della griglia di gioco.

· Altrimenti: applicare algoritmo di Minimax su ogni cella della griglia (sia occupate che libere) aggiungendo ad una lista L i valori di ritorno. A partire da L prendere l'indice dell'elemento che ha valore maggiore (considerando che L sia indicizzato a partire da 1). Questo indice rappresenta la mossa ottima.

  - Aggiornare la griglia di gioco con la nuova mossa acquisita.
  - Passare il turno all'avversario.

- Dichiarare il risultato della partita.

I passi dell'algoritmo di Minimax sono i seguenti:

- Caso base:
  - Se il gioco termina in seguito ad una mossa, allora si ritorna:
    * in caso di vittoria, un valore $V = +P - D$;
    * in caso si sconfitta, un valore $V = -P + D$;
    * in caso di parità, il valore 0;

    dove con P si intende il punteggio di guadagno/perdita $(10/-10)$ e con D il livello di profondità nell'albero delle simulazioni a cui si sta effettuando la valutazione.
  - Se la mossa considerata non fosse valida perché la cella corrispondente risulta essere già occupata si ritorna -1000 o $+1000$ a seconda che il giocatore precedente fosse rispettivamente MAX oppure MIN.
- Caso generale: se il gioco non termina in seguito ad una mossa, allora
  - si cambia il giocatore, passando da MAX a MIN o viceversa;
  - si incrementa di uno la profondità dell'albero;
  - si aggiorna la griglia di gioco con la mossa corrente;
  - su queste nuove condizioni si invoca ricorsivamente l'algoritmo di Minimax per ognuna delle celle nella griglia di gioco.

  Una volta che le nove istanze dell'algoritmo di Minimax che sono state invocate ritornano i loro valori, questi si salvano all'interno di una lista (indicizzata a partire da 1) e si seleziona come valore di ritorno il valore massimo o minimo trovato a seconda che il giocatore corrente sia rispettivamente MAX oppure MIN.

4 Implementazione dell'Algoritmo

Si ritiene opportuno commentare la scelta della rivisitazione dell'indentazione del codice. Lo stile d'indentazione è stato necessariamente rimodellato al fine di rendere il codice leggibile compatibilmente con le dimensioni della pagina.

File sorgente `tic_tac_toe.hs`:

```
{- Programma Haskell che implementa il gioco del Tic-Tac-Toe
   singolo giocatore con algoritmo di ottimiazzazione Minimax.

   Nota: con "Giocatore" si fa riferimento al giocatore pilotato
   dall'utente, mentre con "CPU" ci si riferisce a quello pilotato
   dal computer.

   Nota: nel programma si rappresenta la griglia di gioco ed il suo
   stato attraverso una lista di caratteri. I controlli fatti
   su questa lista non saranno particolarmente stringenti dato che
   questa viene gestita interamente dal programma e quindi si
   garantisce una correttezza intrinseca. -}


{-------------------------------------------------------------------------
 --                        IMPORTAZIONE DEI MODULI                       --
 -------------------------------------------------------------------------}

{- Necessario per usare elemIndex, che restituisce l'indice del primo
   elemento che soddisfa una certa condizione. -}
import Data.List (elemIndex)

{- Necessario per usare inRange, che restituisce True nel caso in cui
   un valore dato si trovi all'interno dei limiti specificati. -}
import Data.Ix (inRange)

{- Necessario per usare fromJust, che estrae l'elemento da un costruttore
   Just. -}
import Data.Maybe (fromJust)

{- Necessario per usare readMaybe, che consente di gestire la validazione
   dell'input. -}
import Text.Read (readMaybe)


{-------------------------------------------------------------------------
 --                      RIDENOMINAZIONE DEI TIPI                        --
 -------------------------------------------------------------------------}

{- Dato che la griglia di gioco viene rappresentata come una lista di
   caratteri mentre una mossa viene rappresentata mediante una cifra,
   si sceglie di effettuare le seguenti ridenominazioni:
   - il tipo strutturato [Char] viene ridenominato in Griglia;
   - il tipo scalare Int viene ridenominato in Mossa. -}

type Griglia = [Char]
type Mossa = Int

{-------------------------------------------------------------------------
 --                      INIZIALIZZAZIONE DEL GIOCO                      --
 -------------------------------------------------------------------------}

main :: IO ()
main =
```

```
    do putStrLn "Gioco del Tic-Tac-Toe!\n"
       gioca


{- La funzione gioca inizializza l'interazione con il Giocatore
   acquisendo un carattere:
    - 's' allora si inizia una nuova partita;
    - 'h' vengono mostrate le regole al giocatore.
   L'uso di _ <- getLine consente la pulizia del buffer. -}

gioca :: IO ()
gioca =
    do putStr   "Digita 's' per iniziare a giocare oppure 'h' per"
       putStrLn " consultare le regole:"
       c <- getChar
       _ <- getLine
       inizio_gioco c


{- La funzione inizio_gioco gestisce la richiesta del Giocatore:
   - l'argomento è il comando dato dal Giocatore.
   La partita favorisce sempre il Giocatore che quindi partirà per primo. -}

inizio_gioco :: Char -> IO ()
inizio_gioco c
    | c == 's'  =
        do partita [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '] 1 1
    | c == 'h'  =
        do mostra_regole
           gioca
    | otherwise =
        do putStrLn $ "Input " ++ show [c] ++ " non valido."
           gioca


{-------------------------------------------------------------------------
 --                IMPLEMENTAZIONE FUNZIONI DI UTILITÀ                  --
 -------------------------------------------------------------------------}

{- La funzione mostra_regole si occupa di mostrare al Giocatore le regole
   del gioco. -}

mostra_regole :: IO ()
mostra_regole =
    do putStrLn "\nREGOLE DEL GIOCO:"
       putStrLn " - il primo turno spetta al Giocatore;"
       putStrLn " - al Giocatore viene assegnato il simbolo 'o';"
       putStrLn " - a CPU viene assegnato il simbolo 'x';"
       putStr   " - vince chi riesce a disporre tre dei propri simboli in"
       putStrLn " linea retta orizzontale, verticale o diagonale;"
       putStr   " - se la griglia viene riempita senza che nessuno dei"
       putStr   " giocatori sia riuscito a completare una linea retta"
       putStrLn " di tre simboli, il gioco finisce in parità."
       putStrLn "\nCOME FARE UNA MOSSA:"
       putStr   " - fare una mossa vuol dire digitare il numero che"
       putStr   " corrisponde alla cella in cui si vuole aggiungere"
```

```
        putStrLn " il proprio simbolo."
        putStr   "Ad esempio: digitare '5' per aggiungere 'o' nella cella"
        putStrLn " centrale."
        putStrLn "\nCOME SI COMPONE LA GRIGLIA:"
        putStrLn " - la griglia si compone di nove celle;"
        putStrLn " - le celle sono numerate da 1 a 9."
        putStrLn "\nGRIGLIA DI RIFERIMENTO:\n"
        disegna_griglia ['1', '2', '3', '4', '5', '6', '7', '8', '9'] 1


{- La funzione disegna_griglia stampa la griglia di gioco:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è un indice che identifica la cella da disegnare.
   Per stato corrente del gioco si intende l'insieme dei
   simboli attualmente presenti nella griglia. -}

disegna_griglia :: Griglia -> Int -> IO ()
disegna_griglia (s : ls) i
    | i `mod` 3 /= 0     =
        do putStr " "
           putChar s
           putStr " |"
           disegna_griglia ls (i + 1)
    | (i == 3 || i == 6) =
        do putStr " "
           putChar s
           putStr "\n---+---+---\n"
           disegna_griglia ls (i + 1)
    | i == 9             =
        do putStr " "
           putChar s
           putStr "\n\n"


{- La funzione acquisisci_mossa_giocatore ottiene la mossa
   fatta dal Giocatore:
   - l'argomento è la lista contenente lo stato corrente del gioco.
   Il nuovo stato del gioco dipenderà dalla mossa acquisita.
   Si fa notare che alla funzione mossa_valida si passa la mossa
   decrementata di 1 (dato che in Haskell le liste sono indicizzate a
   partire da 0); questo si fa per favorire l'esperienza di gioco
   dell'utente e mantenere la numerazione delle celle da 1 a 9. -}

acquisisci_mossa_giocatore :: Griglia -> IO Mossa
acquisisci_mossa_giocatore l =
    do putStrLn "Digita il numero della cella:"
       input <- getLine
       let m = case readMaybe input :: Maybe Mossa of
                   Just i  -> i
                   Nothing -> -1
       if (mossa_valida l (m-1))
           then return m
           else do putStrLn "Mossa non valida."
                   acquisisci_mossa_giocatore l
```

```
{- La funzione mosse_rimaste verifica che la griglia di gioco non sia
   completamente piena:
   - l'argomento è la lista contenente lo stato corrente del gioco.
   La griglia di gioco si dice piena quando ogni sua cella è occupata
   dal un simbolo ('x' oppure 'o').
   L'uso di any consente di applicare la condizione specificata come
   primo argomento ad ogni elemento della lista (specificata come
   secondo). -}

mosse_rimaste :: Griglia -> Bool
mosse_rimaste l = any (==' ') l


{- La funzione mossa_valida verifica che una mossa sia valida:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la mossa che si vuole validare.
   Una mossa è valida quando la cella corrispondente è libera. -}

mossa_valida :: Griglia -> Mossa -> Bool
mossa_valida l m
    | inRange (0,8) m && l!!m == ' ' = True
    | otherwise                      = False


{- La funzione aggiorna_griglia genera il nuovo stato della griglia di gioco
   inserendo un simbolo ('x' oppure 'o' a seconda del giocatore) nella cella
   specificata:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il numero della cella che si vuole aggiornare;
   - il terzo argomento è il simbolo che si vuole aggiungere. -}

aggiorna_griglia :: Griglia -> Mossa -> Char -> Griglia
aggiorna_griglia (_ : ls) 0 s              = s : ls
aggiorna_griglia (ts : ls) i s | i > 0     = ts : aggiorna_griglia ls (i-1) s
                               | otherwise = []


{- La funzione stampa_risultato_partita stampa il risultato della partita.
   L'argomento indica la codifica del risultato:
   - 0 vuol dire che la partita termina in parità;
   - 1 vuol dire che Giocatore ha vinto;
   - 2 vuol dire che CPU ha vinto. -}

stampa_risultato_partita :: Int -> IO ()
stampa_risultato_partita n | n == 0 = do putStrLn "Pari!"
                           | n == 1 = do putStrLn "Hai vinto!"
                           | n == 2 = do putStrLn "Hai perso, CPU vince!"


{- La funzione controlla_vincitore verifica se nella griglia sia presente
   una configurazione per cui si possa decretare un vincitore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il simbolo per cui si vuole controllare. -}

controlla_vincitore :: Griglia -> Char -> Bool
controlla_vincitore l s  | controlla_righe     = True
```

```
                        | controlla_colonne   = True
                        | controlla_diagonali = True
                        | otherwise           = False
    where
        controlla_righe     = (l!!0 == s && l!!1 == s && l!!2 == s)
                              || (l!!3 == s && l!!4 == s && l!!5 == s)
                              || (l!!6 == s && l!!7 == s && l!!8 == s)
        controlla_colonne   = (l!!0 == s && l!!3 == s && l!!6 == s)
                              || (l!!1 == s && l!!4 == s && l!!7 == s)
                              || (l!!2 == s && l!!5 == s && l!!8 == s)
        controlla_diagonali = (l!!0 == s && l!!4 == s && l!!8 == s)
                              || (l!!2 == s && l!!4 == s && l!!6 == s)


{-------------------------------------------------------------------------
 --                      IMPLEMENTAZIONE LOOP DEL GIOCO                  --
 -------------------------------------------------------------------------}

{- La funzione partita implementa il loop del gioco, gestendo il passsaggio
   del turno, le mosse di Giocatore e CPU, e la terminazione della partita:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento rappresenta il giocatore in possesso del turno;
   - il terzo argomento rappresenta il numero del turno raggiunto. -}

partita :: Griglia -> Int -> Int -> IO ()
partita l p t
    {- Gestione del turno di Giocatore: secondo argomento uguale a 1. -}
    | p == 1 && t < 10 && not(controlla_vincitore l 'x')          =
        do disegna_griglia l 1
           mossa_giocatore <- acquisisci_mossa_giocatore l
           partita (aggiorna_griglia l (mossa_giocatore - 1) 'o') 2 (t + 1)
    | p == 1 && t < 10 && controlla_vincitore l 'x'               =
        do disegna_griglia l 1
           stampa_risultato_partita 2
    {- Gestione del turno di CPU: secondo argomento uguale a 2. -}
    | p == 2 && t > 2 && t < 10 && not(controlla_vincitore l 'o') =
        do disegna_griglia l 1
           let mossa_cpu = cerca_mossa_migliore l
           partita (aggiorna_griglia l mossa_cpu 'x') 1 (t + 1)
    | p == 2 && t > 2 && t < 10 && controlla_vincitore l 'o'      =
        do disegna_griglia l 1
           stampa_risultato_partita 1
    {- Per decidere la prima mossa di CPU non si utilizza l'algoritmo
       di Minimax, bensì le strategie note del Tic-Tac-Toe:
       - se libera, prendere subito la cella centrale della griglia;
       - se la cella centrale è occupata, scegliere uno qualsiasi
         degli angoli della griglia. -}
    | p == 2 && t == 2 && mossa_valida l 4                        =
        do disegna_griglia l 1
           partita (aggiorna_griglia l 4 'x') 1 (t + 1)
    | p == 2 && t == 2 && mossa_valida l 0                        =
        do disegna_griglia l 1
           partita (aggiorna_griglia l 0 'x') 1 (t + 1)
    {- Per decidere il risultato una volta raggiunto il numero massimo
       di mosse: 10. -}
    | t == 10 && controlla_vincitore l 'o'                        =
```

```
        do disegna_griglia l 1
           stampa_risultato_partita 1
    | t == 10 && not (controlla_vincitore l 'o')                    =
        do disegna_griglia l 1
           stampa_risultato_partita 0


{-------------------------------------------------------------------------
 --                  IMPLEMENTAZIONE ALGORITMO DI MINIMAX                --
 -------------------------------------------------------------------------}

{- La funzione cerca_mossa_migliore genera la miglior mossa possibile
   che CPU possa fare:
   - l'argomento è la lista contenente lo stato corrente del gioco.
   Per mossa migliore si intende una mossa che non consenta a Giocatore
   di concludere la partita con una vittoria. -}

cerca_mossa_migliore :: Griglia -> Mossa
cerca_mossa_migliore l = fromJust (elemIndex (maximum rs) rs)
    where
        rs = [if (mossa_valida l i)
                  then minimax (aggiorna_griglia l i 'x') 0 False
                  else -1000
              | i <- [0..8]]


{- La funzione minimax calcola il punteggio migliore ottenuto in seguito
   al compimento di una delle possibili mosse:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la profondità raggiunta durante la generazione
     dell'albero delle simulazioni di gioco;
   - il terzo argomento indica se la ricerca del punteggio migliore deve
     essere fatta dal punto di vista del massimizzatore o del minimizzatore.
   Il punteggio migliore viene cercato all'interno di una lista generata
   ad ogni punto di ramificazione lungo l'albero delle simulazioni di gioco. -}

minimax :: Griglia -> Int -> Bool -> Int
minimax l d _     | p == 10                        = p - d
                  | p == -10                       = p + d
    where
        p = valuta l
        {- La funzione valuta calcola il punteggio ottenuto in seguito alla
           valutazione dello stato corrente della griglia del gioco:
           - l'argomento è la lista contenente lo stato corrente del gioco.
           Il punteggio vale:
           - 10 se la vittoria va a CPU;
           - -10 se la vittoria va a Giocatore;
           - 0 se non si è raggiunta una condizione di vittoria. -}
        valuta :: Griglia -> Int
        valuta l | controlla_vincitore l 'x' = 10
                 | controlla_vincitore l 'o' = -10
                 | otherwise                 = 0
minimax l _ _     | not (mosse_rimaste l)          = 0
minimax l d im    | mosse_rimaste l && im == True  = maximum rsMax
                  | mosse_rimaste l && im == False = minimum rsMin
    where
```

```
rsMax = [if (mossa_valida l i)
             then minimax (aggiorna_griglia l i 'x') (d+1) False
             else -1000
         | i <- [0..8]]
rsMin = [if (mossa_valida l i)
             then minimax (aggiorna_griglia l i 'o') (d+1) True
             else 1000
         | i <- [0..8]]
```

File sorgente `tic_tac_toe.pl`:

```
/* Programma Prolog che implementa il gioco del Tic-Tac-Toe
   singolo giocatore con algoritmo di ottimiazzazione Minimax.

   Nota: con "Giocatore" si fa riferimento al giocatore pilotato
   dall'utente, mentre con "CPU" ci si riferisce a quello pilotato
   dal computer.

   Nota: nel programma si rappresenta la griglia di gioco ed il suo
   stato attraverso una lista di caratteri. I controlli fatti
   su questa lista non saranno particolarmente stringenti dato che
   questa viene gestita interamente dal programma e quindi si
   garantisce una correttezza intrinseca. */


/*************************************************************************
 **                        INIZIALIZZAZIONE DEL GIOCO                   **
 *************************************************************************/

main :-
    write('Gioco del Tic-Tac-Toe!'), nl, nl,
    gioca.


/* Il predicato gioca inizializza l'interazione con il Giocatore
   acquisendo un carattere:
    - 's' allora si inizia una nuova partita;
    - 'h' vengono mostrate le regole al Giocatore. */

gioca :-
    write('Digita \'s\' per iniziare a giocare oppure \'h\' per'),
    write(' consultare le regole:'), nl,
    get_char(C),
    pulisci_input_buffer,
    inizio_gioco(C).


/* Il predicato inizio_gioco gestisce la richiesta del Giocatore:
   - l'argomento è il comando dato dal Giocatore.
   La partita favorisce sempre il Giocatore che quindi partirà per primo. */

inizio_gioco(C) :-
    C = 's',
    partita([' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '], 1, 1).
inizio_gioco(C) :-
    C = 'h',
    mostra_regole,
    gioca.
inizio_gioco(C) :-
    C \= 's', C \= 'h',
    write('Input \"'), write(C), write('\" non valido.'), nl,
    gioca.
```

```
/*************************************************************************
 **                 IMPLEMENTAZIONE PREDICATI DI UTILITÀ                **
 *************************************************************************/

/* Il predicato mostra_regole si occupa di mostrare al Giocatore le regole
   del gioco. */

mostra_regole :-
    nl, write('REGOLE DEL GIOCO:'), nl,
    write(' - il primo turno spetta al Giocatore;'), nl,
    write(' - al Giocatore viene assegnato il simbolo \'o\';'), nl,
    write(' - a CPU viene assegnato il simbolo \'x\';'), nl,
    write(' - vince chi riesce a disporre tre dei propri simboli in linea'),
    write(' retta orizzontale, verticale o diagonale;'), nl,
    write(' - se la griglia viene riempita senza che nessuno dei'),
    write(' giocatori sia riuscito a completare una linea'),
    write(' retta di tre simboli, il gioco finisce in parità.'), nl, nl,
    write('COME FARE UNA MOSSA:'), nl,
    write(' - fare una mossa vuol dire digitare il numero che corrisponde'),
    write(' alla cella in cui si vuole aggiungere il proprio simbolo.'), nl,
    write('Ad esempio: digitare \'5\' per aggiungere \'o\' nella cella'),
    write(' centrale.'), nl, nl,
    write('COME SI COMPONE LA GRIGLIA:'), nl,
    write(' - la griglia si compone di nove celle;'), nl,
    write(' - le celle sono numerate da 1 a 9.'), nl, nl,
    write('GRIGLIA DI RIFERIMENTO:'), nl, nl,
    disegna_griglia(['1', '2', '3', '4', '5', '6', '7', '8', '9'], 1).


/* Il predicato disegna_griglia stampa la griglia di gioco:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è un indice che identifica la cella da disegnare.
   Per stato corrente del gioco si intende l'insieme dei
   simboli attualmente presenti nella griglia.
   L'uso del predicato nth permette di ottenere l'elemento all'indice
   specificato come primo argomento, della lista specificata come secondo
   argomento. */

disegna_griglia(L, I) :-
    mod(I, 3) =\= 0,
    nth(I, L, Y),
    write(' '), write(Y), write(' |'),
    J is I + 1,
    disegna_griglia(L, J).
disegna_griglia(L, I) :-
    (I = 3 ; I = 6),
    nth(I, L, Y),
    write(' '), write(Y), nl, write('---+---+---'), nl,
    J is I + 1,
    disegna_griglia(L, J).
disegna_griglia(L, I) :-
    I = 9,
    nth(I, L, Y),
    write(' '), write(Y), nl, nl, !.
```

```
/* Il predicato acquisisci_mossa_giocatore ottiene la mossa
   fatta dal Giocatore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la lista contenente il nuovo stato del gioco.
   Il nuovo stato del gioco dipenderà dalla mossa acquisita.
   Il predicato catch permette di implementare il costrutto try-catch. */

acquisisci_mossa_giocatore(L, NL) :-
    write('Digita il numero della cella:'), nl,
    catch(
        (    read_integer(M),
             (    mossa_valida(L, M) *->
                  aggiorna_griglia(L, M, 'o', NL)
             ;    throw(_)
             )
         ),
         error(_, _),
         (    write('Mossa non valida.'), nl,
              pulisci_input_buffer,
              acquisisci_mossa_giocatore(L, NL)
         )
    ).


/* Il predicato mossa_cpu aggiorna la griglia di gioco aggiungendo
   la mossa fatta da CPU:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la lista contenente lo stato
     aggiornato del gioco. */

mossa_cpu(L, NL) :-
    cerca_mossa_migliore(L, MM),
    aggiorna_griglia(L, MM, 'x', NL).


/* Il predicato pulisci_input_buffer pulisce il buffer rimuovendo i
   caratteri in eccesso.
   La pulizia non termina finché non si legge un carattere
   con codice 10 (carattere di fine linea). */

pulisci_input_buffer :- repeat, get_code(I), I = 10, !.


/* Il predicato mosse_rimaste verifica che la griglia di gioco non sia
   completamente piena:
   - l'argomento è la lista contenente lo stato corrente del gioco.
   La griglia di gioco si dice piena quando ogni sua cella è occupata
   dal un simbolo ('x' oppure 'o').
   Il predicato between ha successo se il terzo argomento è
   compreso tra i primi due (inclusi). */

mosse_rimaste(L) :- between(1, 9, I), mossa_valida(L, I), !.


/* Il predicato mossa_valida verifica che una mossa sia valida:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
```

```
   - il secondo argomento è la mossa che si vuole validare.
   Una mossa è valida quando la cella corrispondente è libera. */

mossa_valida(L, M) :- nth(M, L, ' ').


/* Il predicato aggiorna_griglia genera il nuovo stato della griglia di
   gioco inserendo un simbolo ('x' oppure 'o' a seconda del giocatore)
   nella cella specificata:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il numero della cella che si vuole aggiornare;
   - il terzo argomento è il simbolo che si vuole aggiungere;
   - il quarto argomento è la lista contenente lo stato aggiornato
     del gioco. */

aggiorna_griglia([_|LS], 1, S, [S|LS]).
aggiorna_griglia([TS|LS], I, S, [TS|TLS]) :-
    I > 1, I1 is I-1,
    aggiorna_griglia(LS, I1, S, TLS).


/* Il predicato stampa_risultato_partita stampa il risultato della partita.
   L'argomento indica la codifica del risultato:
   - 0 vuol dire che la partita termina in parità;
   - 1 vuol dire che Giocatore ha vinto;
   - 2 vuol dire che CPU ha vinto. */

stampa_risultato_partita(0) :- write('Pari!'), nl.
stampa_risultato_partita(1) :- write('Hai vinto!'), nl.
stampa_risultato_partita(2) :- write('Hai perso, CPU vince!'), nl.


/* Il predicato controlla_vincitore verifica se nella griglia sia presente
   una configurazione per cui si possa decretare un vincitore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il simbolo per cui si vuole controllare. */

controlla_vincitore(L, S) :- controlla_righe(L, S).
controlla_vincitore(L, S) :- controlla_colonne(L, S).
controlla_vincitore(L, S) :- controlla_diagonali(L, S).


/* Il predicato controlla_righe verifica se nella griglia sia presente
   una configurazione orizzontale per cui si possa decretare un vincitore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il simbolo per cui si vuole controllare. */

controlla_righe([S1, S2, S3, _, _, _, _, _, _], I) :- S1 = I, S2 = I, S3 = I.
controlla_righe([_, _, _, S4, S5, S6, _, _, _], I) :- S4 = I, S5 = I, S6 = I.
controlla_righe([_, _, _, _, _, _, S7, S8, S9], I) :- S7 = I, S8 = I, S9 = I.


/* Il predicato controlla_colonne verifica se nella griglia sia presente
   una configurazione verticale per cui si possa decretare un vincitore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il simbolo per cui si vuole controllare. */
```

```
controlla_colonne([S1, _, _, S4, _, _, S7, _, _], I) :- S1 = I, S4 = I, S7 = I.
controlla_colonne([_, S2, _, _, S5, _, _, S8, _], I) :- S2 = I, S5 = I, S8 = I.
controlla_colonne([_, _, S3, _, _, S6, _, _, S9], I) :- S3 = I, S6 = I, S9 = I.


/* Il predicato controlla_diagonali verifica se nella griglia sia presente
   una configurazione diagonale per cui si possa decretare un vincitore:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il simbolo per cui si vuole controllare. */

controlla_diagonali([S1, _, _, _, S5, _, _, _, S9], I) :- S1 = I, S5 = I, S9 = I.
controlla_diagonali([_, _, S3, _, S5, _, S7, _, _], I) :- S3 = I, S5 = I, S7 = I.


/*************************************************************************
 **                      IMPLEMENTAZIONE LOOP DEL GIOCO                 **
 *************************************************************************/

/* Il predicato partita implementa il loop del gioco, gestendo il passsaggio
   del turno, le mosse di Giocatore e CPU, e la terminazione della partita:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento rappresenta il giocatore in possesso del turno;
   - il terzo argomento rappresenta il numero del turno raggiunto. */

/* Gestione del turno di Giocatore: secondo argomento uguale a 1. */
partita(L, 1, T)  :-
    T < 10,
    \+(controlla_vincitore(L, 'x')),
    disegna_griglia(L, 1),
    acquisisci_mossa_giocatore(L, NL),
    NT is T + 1,
    partita(NL, 2, NT).
partita(L, 1, T)  :-
    T < 10,
    controlla_vincitore(L, 'x'),
    disegna_griglia(L, 1),
    stampa_risultato_partita(2), !.
/* Gestione del turno di CPU: secondo argomento uguale a 2. */
partita(L, 2, T)  :-
    T > 2, T < 10,
    \+(controlla_vincitore(L, 'o')),
    disegna_griglia(L, 1),
    mossa_cpu(L, NL),
    NT is T + 1,
    partita(NL, 1, NT).
partita(L, 2, T)  :-
    T > 2, T < 10,
    controlla_vincitore(L, 'o'),
    disegna_griglia(L, 1),
    stampa_risultato_partita(1), !.
/* Per decidere la prima mossa di CPU non si utilizza l'algoritmo di Minimax,
   bensì le strategie note del Tic-Tac-Toe:
   - se libera, prendere subito la cella centrale della griglia;
   - se la cella centrale è occupata, scegliere uno qualsiasi degli angoli
     della griglia. */
```

```
partita(L, 2, T)  :-
    T =:= 2,
    mossa_valida(L, 5),
    aggiorna_griglia(L, 5, 'x', NL),
    disegna_griglia(L, 1),
    NT is T + 1,
    partita(NL, 1, NT).
partita(L, 2, T)  :-
    T =:= 2,
    mossa_valida(L, 1),
    aggiorna_griglia(L, 1, 'x', NL),
    disegna_griglia(L, 1),
    NT is T + 1,
    partita(NL, 1, NT).
/* Per decidere il risultato una volta raggiunto il numero massimo
   di mosse: 10. */
partita(L, _, 10) :-
    controlla_vincitore(L, 'o'),
    disegna_griglia(L, 1),
    stampa_risultato_partita(1), !.
partita(L, _, 10) :-
    \+(controlla_vincitore(L, 'o')),
    disegna_griglia(L, 1),
    stampa_risultato_partita(0), !.


/*************************************************************************
 **                   IMPLEMENTAZIONE ALGORITMO DI MINIMAX                **
 *************************************************************************/

/* Il predicato cerca_mossa_migliore genera la miglior mossa possibile
   che CPU possa fare:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la mossa migliore possibile.
   Per mossa migliore si intende una mossa che non consenta a Giocatore
   di concludere la partita con una vittoria.
   Il predicato findall consente di unificare in LR la lista (anche con
   duplicati) contenente tutte le istanze dei risultati R per i quali ha
   successo l'obbiettivo specificato come secondo argomento. */

cerca_mossa_migliore(L, BM) :-
    findall(
         R,
         (    between(1, 9, I),
              (    mossa_valida(L, I),
                   aggiorna_griglia(L, I, 'x', NL),
                   minimax(NL, 0, 0, R) ->
                   true
              ;    R is -1000
              )
         ),
         LR
    ),
    max_list(LR, V),
    nth(BM, LR, V).
```

```
/* Il predicato minimax calcola il punteggio migliore ottenuto in seguito
   al compimento di una delle possibili mosse:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è la profondità raggiunta durante la generazione
     dell'albero delle simulazioni di gioco;
   - il terzo argomento indica se la ricerca del punteggio migliore deve
     essere fatta dal punto di vista del massimizzatore o del minimizzatore;
   - il quarto argomento è il punteggio migliore ottenuto.
   Il punteggio migliore viene cercato all'interno di una lista generata
   ad ogni punto di ramificazione lungo l'albero delle simulazioni di gioco. */

minimax(L, D, _,  BR) :- valuta(L, P), P =:= 10, BR is P - D, !.
minimax(L, D, _,  BR) :- valuta(L, P), P =:= (-10), BR is P + D, !.
minimax(L, _, _,  BR) :- \+(mosse_rimaste(L)), BR is 0, !.
minimax(L, D, IM, BR) :-
    mosse_rimaste(L),
    (    IM = 1 *->
         (    ND is D + 1,
              findall(
                  R,
                  (    between(1, 9, I),
                       (    mossa_valida(L, I),
                            aggiorna_griglia(L, I, 'x', NL),
                            minimax(NL, ND, 0, R) ->
                            true
                       ;    R is -1000
                       )
                  ),
                  LR
              ),
              max_list(LR, BR)
         )
    ;    (    ND is D + 1,
              findall(
                  R,
                  (    between(1, 9, I),
                      (    mossa_valida(L, I),
                           aggiorna_griglia(L, I, 'o', NL),
                           minimax(NL, ND, 1, R) ->
                           true
                       ;    R is 1000
                       )
                  ),
                  LR
              ),
              min_list(LR, BR)
         )
    ), !.


/* Il predicato valuta calcola il punteggio ottenuto in seguito alla
   valutazione dello stato corrente della griglia del gioco:
   - il primo argomento è la lista contenente lo stato corrente del gioco;
   - il secondo argomento è il punteggio.
   Il punteggio vale:
```

```
   - 10 se la vittoria va a CPU;
   - -10 se la vittoria va a Giocatore;
   - 0 se non si è raggiunta una condizione di vittoria. */

valuta(L, P) :- controlla_vincitore(L, 'x'), P is 10.
valuta(L, P) :- controlla_vincitore(L, 'o'), P is (-10).
valuta(L, P) :-
    \+(controlla_vincitore(L, 'x')),
    \+(controlla_vincitore(L, 'o')),
    P is 0.
```

5 Testing del Programma

Test Haskell 1

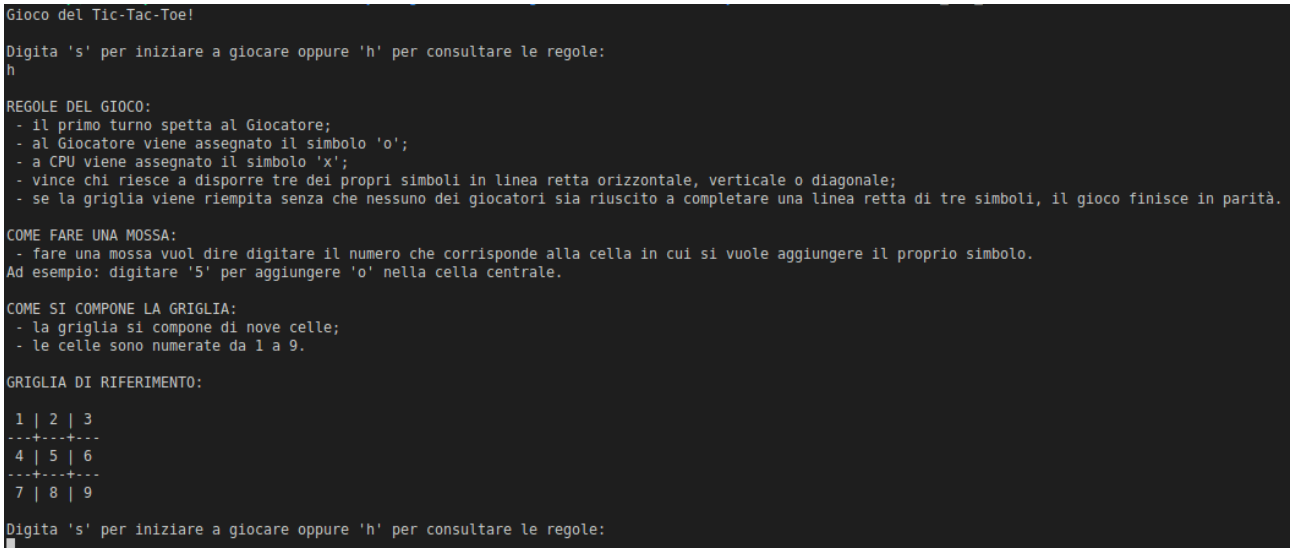

Figura 1: Verifica della corretta stampa del regolamento di gioco.

Test Haskell 2

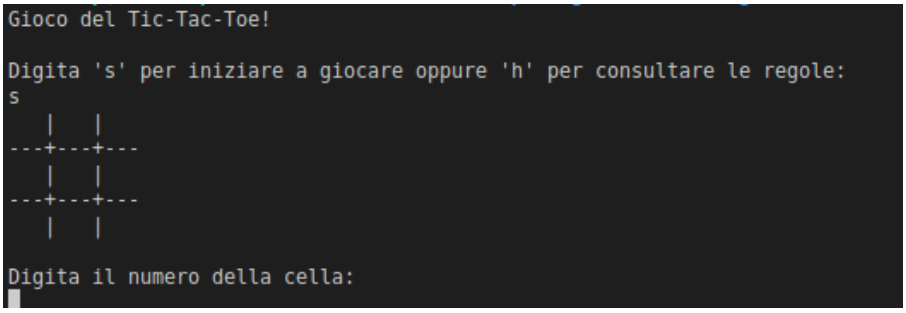

Figura 2: Verifica della corretta inizializzazione del gioco.

Test Haskell 3

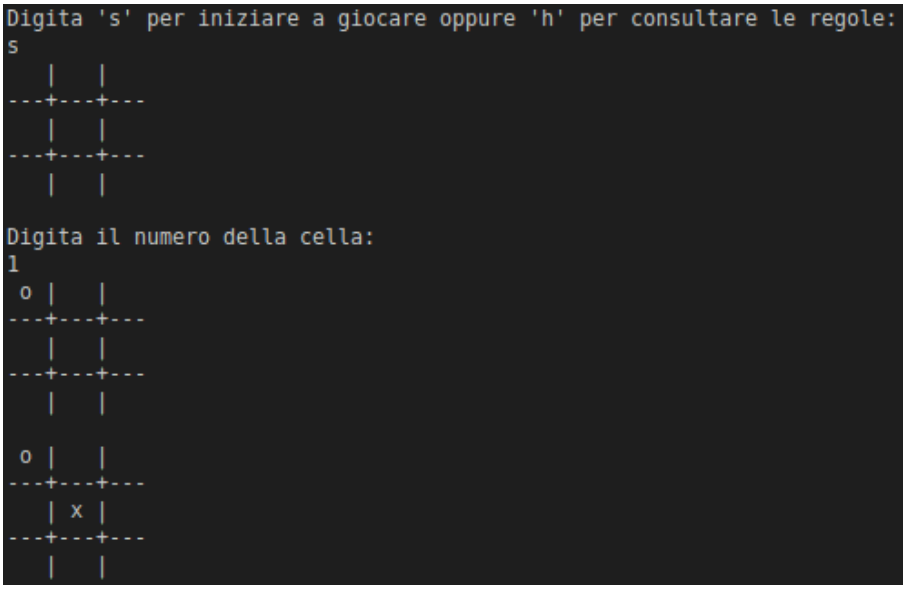

Figura 3: Verifica della scelta della mossa ottima al primo turno giocato da CPU.

Test Haskell 4

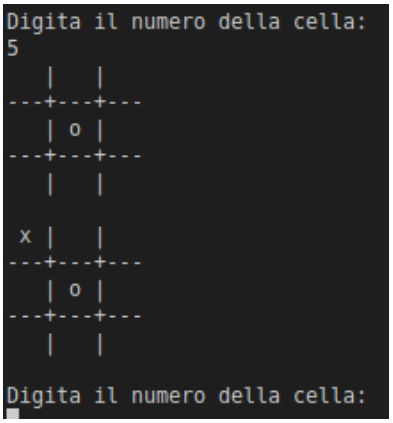

Figura 4: Verifica della scelta della mossa ottima al primo turno giocato da CPU in caso di cella centrale occupata.

Test Haskell 5

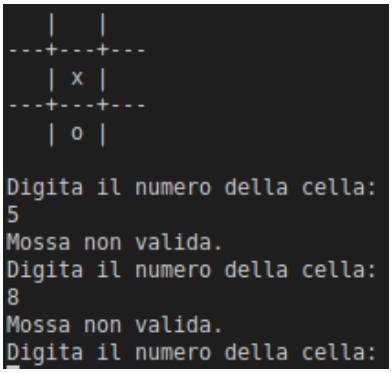

Figura 5: Verifica della correttezza dell'acquisizione della mossa di Giocatore: non è permesso fare una mossa su una cella già occupata.

Test Haskell 6

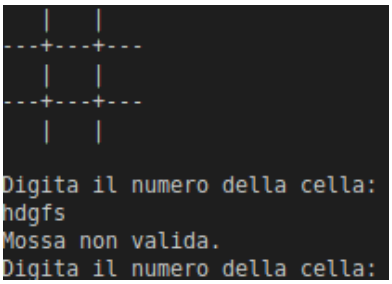

Figura 6: Verifica della correttezza dell'acquisizione della mossa di Giocatore: il formato della mossa non rispecchia quello specificato dal gioco.

Test Haskell 7

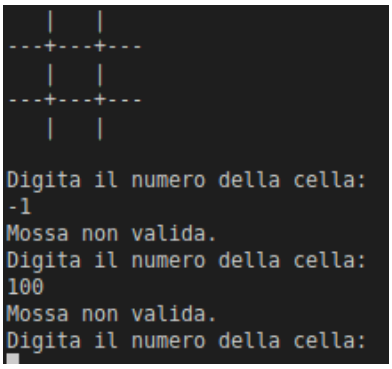

Figura 7: Verifica della correttezza dell'acquisizione della mossa di Giocatore: non si può fare una mossa all'infuori della griglia, per cui questa deve essere compresa tra 1 e 9.

Test Haskell 8

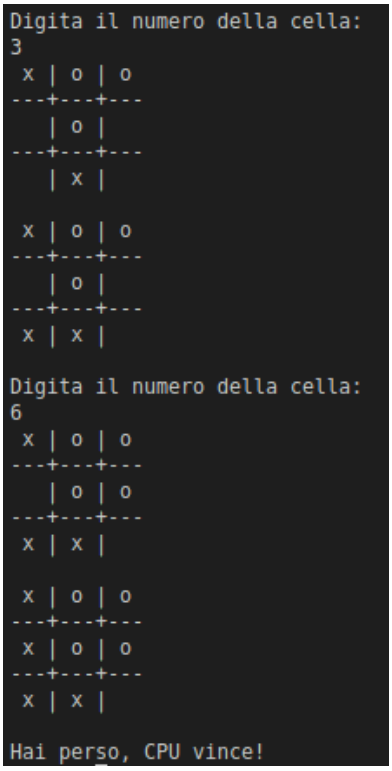

Figura 8: Verifica di correttezza per l'algoritmo di Minimax: CPU fa la mossa ottima attesa vincendo anziché pareggiare posizionando la sua mossa nella cella 9.

Test Haskell 9

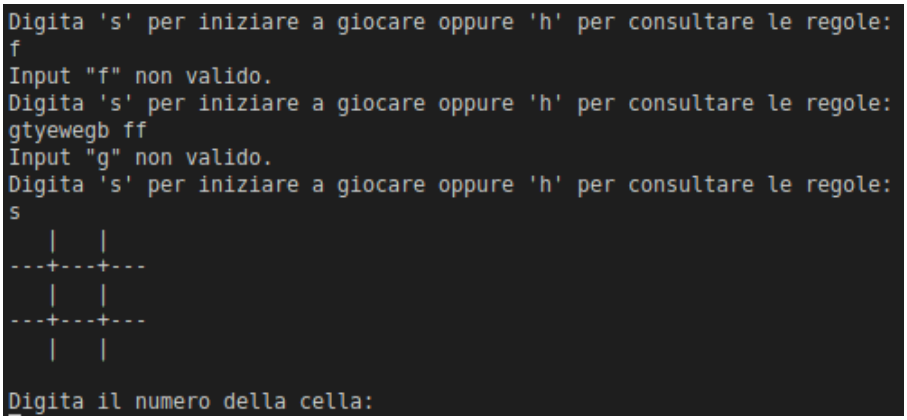

Figura 9: Verifica di correttezza sulla fase validazione dell'input dell'utente.

Test Haskell 10

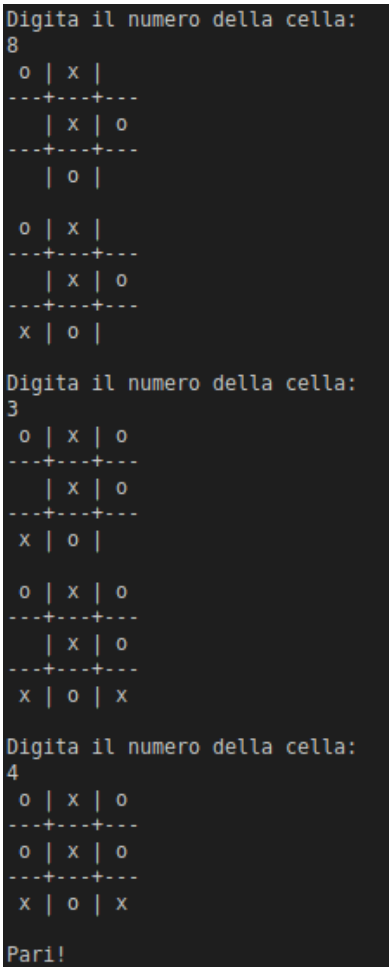

Figura 10: Verifica di correttezza per l'algoritmo di Minimax: nel momento in cui Giocatore fa la sua mossa nella cella 8, più volte si è visto (in fase di sperimentazione) come CPU rispondesse con una mossa nella cella 3 dando la possibilità a Giocatore di vincere.

Test Prolog 1

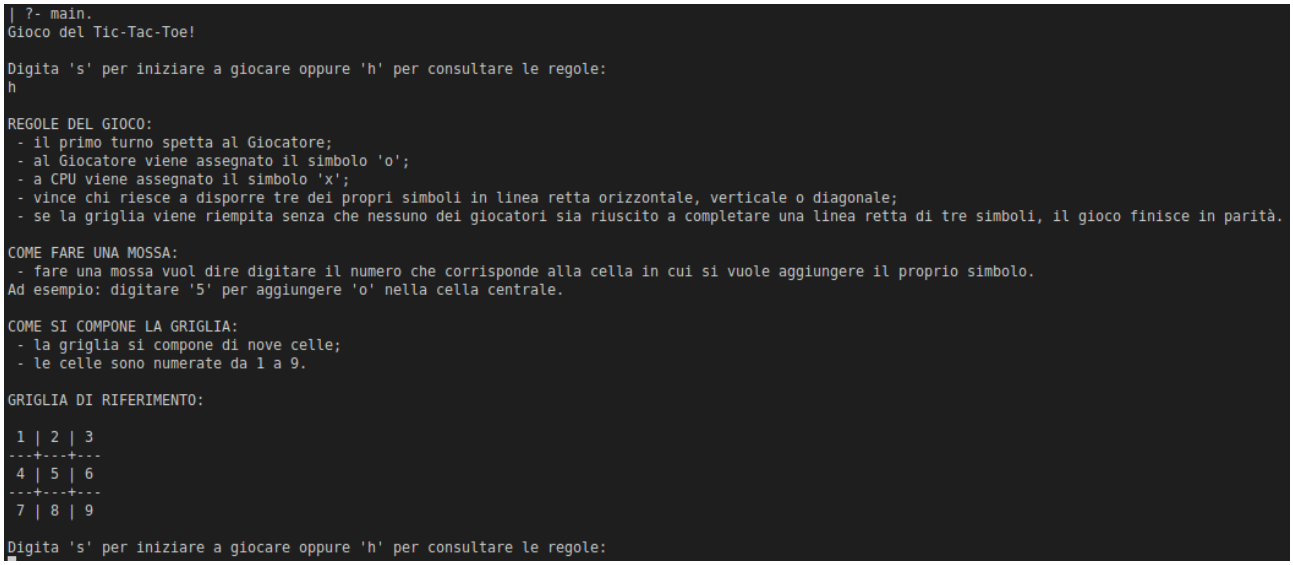

Figura 11: Verifica della corretta stampa del regolamento di gioco.

Test Prolog 2

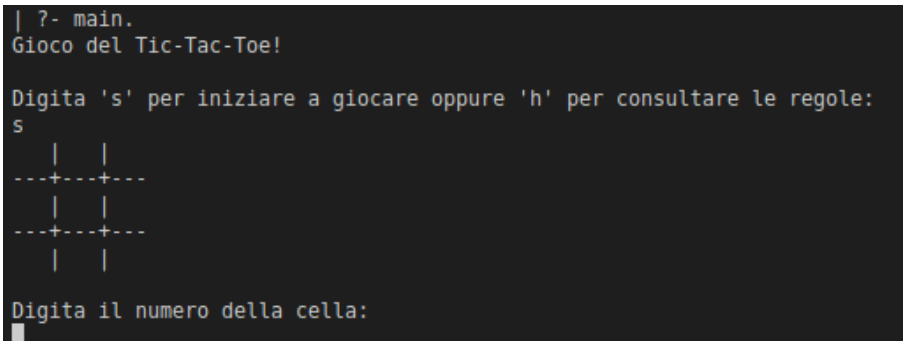

Figura 12: Verifica della corretta inizializzazione del gioco.

Test Prolog 3

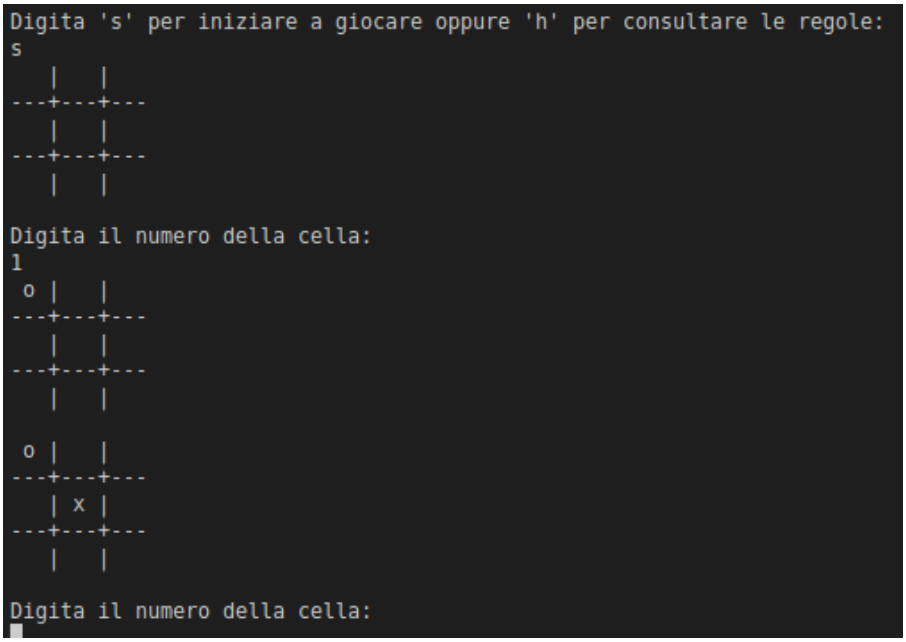

Figura 13: Verifica della scelta della mossa ottima al primo turno giocato da CPU.

Test Prolog 4

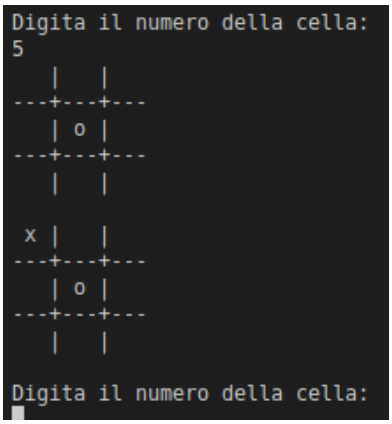

Figura 14: Verifica della scelta della mossa ottima al primo turno giocato da CPU in caso di cella centrale occupata.

Test Prolog 5

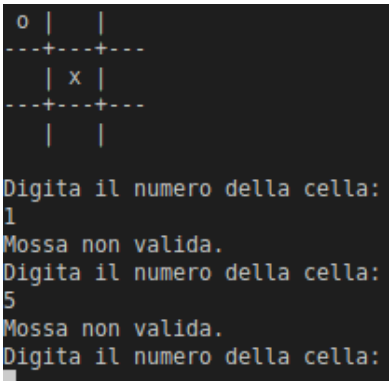

Figura 15: Verifica della correttezza dell'acquisizione della mossa di Giocatore: non è permesso fare una mossa su una cella già occupata.

Test Prolog 6

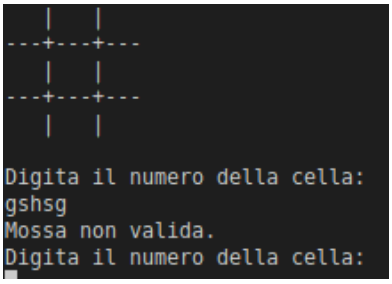

Figura 16: Verifica della correttezza dell'acquisizione della mossa di Giocatore: il formato della mossa non rispecchia quello specificato dal gioco.

Test Prolog 7

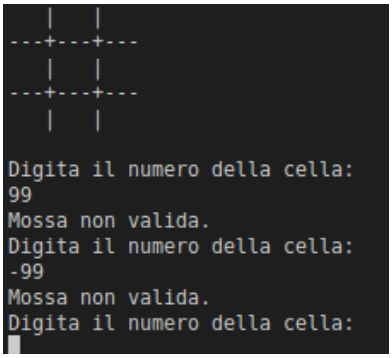

Figura 17: Verifica della correttezza dell'acquisizione della mossa di Giocatore: non si può fare una mossa all'infuori della griglia, per cui questa deve essere compresa tra 1 e 9.

Test Prolog 8

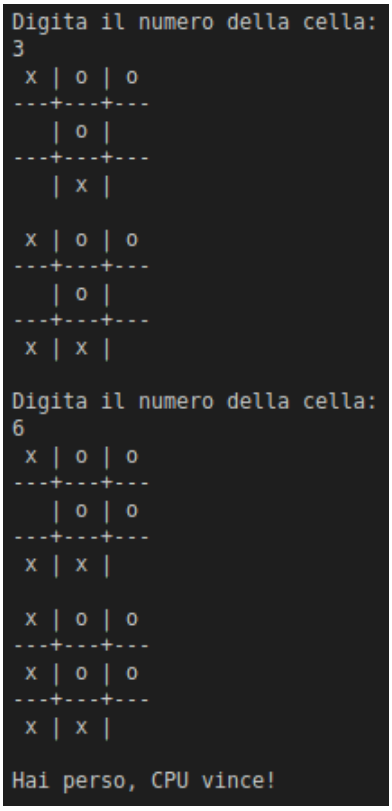

Figura 18: Verifica di correttezza per l'algoritmo di Minimax: CPU fa la mossa ottima attesa vincendo anziché pareggiare posizionando la sua mossa nella cella 9.

Test Prolog 9

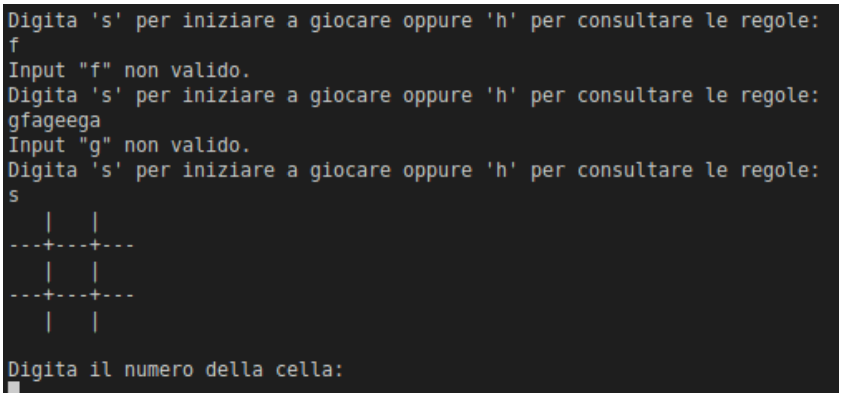

Figura 19: Verifica di correttezza sulla fase validazione dell'input dell'utente.

Test Prolog 10

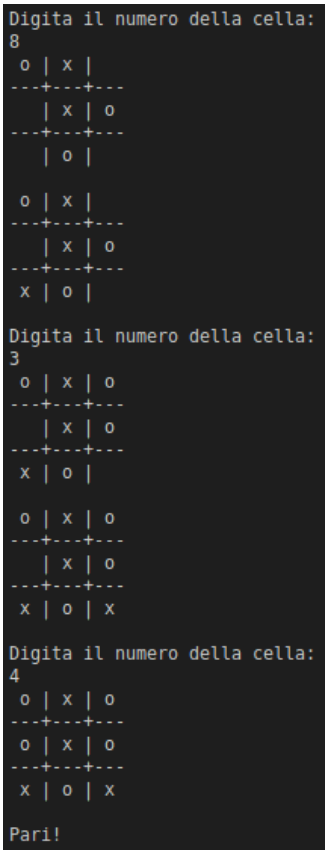

Figura 20: Verifica di correttezza per l'algoritmo di Minimax: nel momento in cui Giocatore fa la sua mossa nella cella 8, più volte si è visto (in fase di sperimentazione) come CPU rispondesse con una mossa nella cella 3 dando la possibilità a Giocatore di vincere.