A functional program to play 3-D Noughts-and-Crosses

Miranda is a functional programming language that was developed by David Turner in 1985. It illustrates **literate programming** [see e.g. Wikipedia]. Literate programming [conceived by Don Knuth, 1981] is a philosophy of computer programming based on the notion that a computer program should be written as if it were a piece of literature, with human readability as a primary goal. Literate programming is achieved by combining human-readable documentation and machine-readable source code into a single source file, in order to maintain a close correspondence between documentation and source code. In the Miranda 3D OXO program, the machine-readable source is prefixed by ">" symbols.

Some commentary on the program by Beynon is given in italics. This draws attention to issues relating to EM that are topical in connection with oxoJoy1994. The non-italised text is due to Mike Joy, who developed the program, and who also played a key role in constructing the EM model oxoJoy1994.

```
||MIRANDA 3-Dimensional Noughts-and-Crosses (author: Mike Joy)

This program plays noughts and crosses in 3-dimensions.

> %export oxo

> inputdatum ::= Prompt | Error | NodeNumber num
```

The `inputdatum` corresponds to a transaction made by the computer - outputing a prompt, an error message or a move. Note that all of these transactions have a similar effect, transforming a string that represents the entire history of the game so far into a string that represents the history of the game so far after a single move has been made (if legal) or ventured (if illegal). The string that represents the history of the game so far includes as sub-segments representations of all the board positions to date by strings of characters, separated by messages such as prompts, error and game status messages. Note the way that this approach abstracts away what - from an EM perspective - are very significant distinctions between observables such as board positions and error messages, and also abstracts away state change and agency.

The square weightings are used when deciding how to evaluate particular squares before making a move.

```
square weightings for simple algorithm

> crossvals :: [num]
> crossvals = [1,6,36,10000]
> noughtvals :: [num]
> noughtvals = [1,4,16,200]
```

The next piece of the code describes the 4-by-4-by-4 OXO board. It first describes the start position, but must subsequently model the stream of positions that arise in a game.

```
startboard is the initial position.

> startboard :: [char]
> startboard = rep 64 '.'

rows is a list of 4-tuples representing the possible lines on the board.

> rows :: [[num]]
> rows = [[0,1,2,3], [0,4,8,12], [0,5,10,15], [1,5,9,13], [2,6,10,14],
>                [3,6,9,12], [3,7,11,15], [4,5,6,7], [8,9,10,11], [12,13,14,15],
>                [0,16,32,48], [0,17,34,51], [1,17,33,49], [2,18,34,50],
>                [3,18,33,48], [3,19,35,51], [0,20,40,60], [0,21,42,63],
>                [1,21,41,61], [2,22,42,62], [3,22,41,60], [3,23,43,63],
>                [4,20,36,52], [4,21,38,55], [5,21,37,53], [6,22,38,54],
```

```
>               [7,22,37,52], [7,23,39,55], [8,24,40,56], [8,25,42,59],
>               [9,25,41,57], [10,26,42,58], [11,26,41,56], [11,27,43,59],
>               [12,24,36,48], [12,25,38,51], [13,25,37,49], [14,26,38,50],
>               [15,26,37,48], [15,27,39,51], [12,28,44,60], [12,29,46,63],
>               [13,29,45,61], [14,30,46,62], [15,30,45,60], [15,31,47,63],
>               [16,17,18,19], [16,20,24,28], [16,21,26,31], [17,21,25,29],
>               [18,22,26,30], [19,22,25,28], [19,23,27,31], [20,21,22,23],
>               [24,25,26,27], [28,29,30,31], [32,33,34,35], [32,36,40,44],
>               [32,37,42,47], [33,37,41,45], [34,38,42,46], [35,38,41,44],
>               [35,39,43,47], [36,37,38,39], [40,41,42,43], [44,45,46,47],
>               [48,49,50,51], [48,52,56,60], [48,53,58,63], [49,53,57,61],
>               [50,54,58,62], [51,54,57,60], [51,55,59,63], [52,53,54,55],
>               [56,57,58,59], [60,61,62,63]]
```

nodes is an array of indices of rows corresponding to each square

```
> nodes :: [[num]]
> nodes = [[0,1,2,10,11,16,17], [0,3,12,18], [0,4,13,19], [0,5,6,14,15,20,21],
>               [1,7,22,23], [2,3,7,24], [4,5,7,25], [6,7,26,27], [1,8,28,29],
>               [3,5,8,30], [2,4,8,31], [6,8,32,33], [1,5,9,34,35,40,41],
>               [3,9,36,42], [4,9,37,43], [2,6,9,38,39,44,45], [10,46,47,48],
>               [11,12,46,49], [13,14,46,50], [15,46,51,52], [16,22,47,53],
>               [17,18,23,24,48,49,53],
>               [19,20,25,26,50,51,53], [21,27,52,53], [28,34,47,54],
>               [29,30,35,36,49,51,54], [31,32,37,38,48,50,54], [33,39,52,54],
>               [40,47,51,55], [41,42,49,55], [43,44,50,55], [45,48,52,55],
>               [10,56,57,58], [12,14,56,49], [11,13,56,60], [15,56,61,62],
>               [22,34,57,63], [24,26,36,38,58,59,63], [23,25,35,37,60,61,63],
>               [27,39,62,63], [16,28,57,64], [18,20,30,32,59,61,64],
>               [17,19,29,31,58,60,64], [21,33,62,64], [40,57,61,65],
>               [42,44,59,65], [41,43,60,65], [45,58,62,65],
>               [10,14,34,38,66,67,68], [12,36,66,69], [13,37,66,70],
>               [11,15,35,39,66,71,72], [22,26,67,73], [24,68,69,73],
>               [25,70,71,73], [23,27,72,73], [28,32,67,74], [30,69,71,74],
>               [31,68,70,74], [29,33,72,74], [16,20,40,44,67,71,75],
>               [18,42,69,75], [19,43,70,75], [17,21,41,45,68,72,75]]
```

The nodes structure holds similar information to the set of `linesthru` observables in oxoJoy1994. The use of the terms 'row' and 'board' in the human-readable and machine-readable code is quite loose.

```
> promptmessage :: [char]
> promptmessage = "Board, column, row? "

> filt :: [char]->num->num->[inputdatum]
> filt [] 0 m = [NodeNumber (m-21)]
> filt [] n m = []
> filt x 0 m = NodeNumber (m-21) : Prompt : filt x 3 0
> filt ('\n' : b) n m  = filt b n m
> filt (' ' : b) n m  = filt b n m
> filt (a : b) n m
>       = Error : Prompt : filt (remrestofline b) 3 0, if ~ (digit a)
>       = Error : Prompt : filt (remrestofline b) 3 0, if code a < code '1'  \/ code a >
>       = NodeNumber (m-21) : Prompt : filt (a:b) 3 0, if n = 0
>       = filt b 2 (4*(code a - code '0')), if n = 3
>       = filt b 1 (m + (code a - code '0')), if n = 2
>       = filt b 0 (m + 16*(code a - code '0')), if n = 1
>       = Error : Prompt : filt (remrestofline b) 3 0, otherwise
```

remrestofline ignores all characters to newline

```
> remrestofline :: [char]->[char]
> remrestofline [] = []
> remrestofline ('\n' : b) = b
> remrestofline (a : b) = remrestofline b
```

`filt` is a function that determines what kind of response is required given a particular board position and

*a projected move. This response is of type "list of `inputdatum`" that abstractly represents the entire sequence of future transformations of the-string-that-represents-the-history-of-the-game-so-far into strings that will represent that history as the game unfolds. To make this work interactively as a game, it is essential to use a **non-strict** (also known as 'lazy') evaluation strategy, whereby the function gives output as it is determined computationally, rather than waiting for the entire sequence of inputs before presenting any output. The stream of inputs to `filt` is supplied by the player on the command line (see below).*

This is the function that determines the response to the current board position given the current input ... this response is to generate appropriate messages together with the next board (if this is appropriate i.e. the player hasn't proposed an invalid move, or the game has just ended).

```
> oxo2 :: [inputdatum]->[char]->[char]
> oxo2 [] bd = "\nYou resign - I win!\n"
> oxo2 (Error : x) bd = "Bad Input\n" ++ oxo2 x bd
> oxo2 (Prompt : x) bd = promptmessage ++ oxo2 x bd
> oxo2 (NodeNumber n : x) bd
>       = "Occupied; try again: " ++ oxo2 x bd, if occupied bd n
>       = "You win; well done!\n", if hlo ~= []
>       = "Drawn game\n", if qm < 0
>       = printboard nb ++showresult qm ++ "\n" ++
>               printboard nbx ++ again, otherwise
>                       where
>                       nb = newboard 'O' n bd
>                       qm = quickmove nb
>                       hlo = hasline 'O' rows nb
>                       hlx = hasline 'X' rows nbx
>                       nbx = newboard 'X' qm nb
>                       again
>                               = "I win, tough luck!\n", if hlx ~= []
>                               = oxo2 x nbx, otherwise

> showresult :: num->[char]
> showresult n
>       = "Draw", if n = -1
>       = "I move " ++ show (((n div 4) mod 4) + 1) ++
>                       show ((n mod 4) + 1) ++
>                       show (n div 16 + 1), otherwise

> oxo1 :: [char]->[char]
> oxo1 = oxo2 (Prompt : filt (read "/dev/tty") 3 0)

newboard takes a board and replaces it with a board where position is
changed to be owned by player.

> newboard :: char->num->[char]->[char]
> newboard player 0 (h:t) = player : t
> newboard player posn (h:t) = h : newboard player (posn-1) t
```

As remarked above: the stream of inputs to `filt` is supplied by the player on the command line. Note that in order to understand how `newboard` represents the switch of player from 'O' to 'X' we have to consider the recursive definition of `newboard` in conjunction with the calls of `newboard` in the definition of the function `oxo2`. In this context, the expression `(h:t)` denotes a list of characters of which `h` is the head and `t` the tail. Figuring out the way this works without the benefit of a Miranda interpreter and a strong grasp of the Miranda syntax is difficult. The next set of functions is concerned with automated play.

```
> linevalue :: [num]->[char]->num
> linevalue x bd
>       = lv2 (lv1 (lv0 x))
>               where
>               lv0 [a,b,c,d] = [bd!a, bd!b, bd!c, bd!d]
>               lv1 [] = (0,0)
```

```
>               lv1 ('X':t) = xadd (lv1 t)
>                             where
>                             xadd (x,o) = (x+1,o)
>               lv1 ('O':t) = oadd (lv1 t)
>                             where
>                             oadd (x,o) = (x,o+1)
>               lv1 (h:t) = lv1 t
>               lv2 (x,0) = crossvals!x
>               lv2 (0,o) = noughtvals!o
>               lv2 (x,o) = 0

> squarevalue :: [char]->num->num
> squarevalue bd n
>       = -1, if bd!n ~= '.'
>       = addv bd (nodes!n), otherwise
>               where
>               addv bd [] = 0
>               addv bd (x:y) = linevalue (rows!x) bd + addv bd y

> values bd = map (squarevalue bd) [0..63]
> quickmove bd
>       = -2, if hasline 'O' rows bd ~= []
>       = -3, if hasline 'X' rows bd ~= []
>       = biggest (values bd) (-1) (-1) 0, otherwise

biggest - arg1 is the list of values, arg2 is the current biggest
value found, arg3 is the square it was found in, arg4 is
the current square being examined.

> biggest :: [num]->num->num->num->num
> biggest [] (-1) square posn = -1
> biggest [] n square posn = square
> biggest (a:b) n square posn
>       = biggest b n square (posn+1), if n > a
>       = biggest b a posn (posn+1), otherwise

> oxo :: [char]
> oxo = "Three dimensional noughts and crosses\n" ++
>               "You are O, I am X, you move first\n" ++
>               printboard startboard ++
>               (oxo1 startboard)

> hasline :: char->[[num]]->[char]->[num]
> hasline plr [] bd = []
> hasline plr (a:b) bd
>       = a, if bd!(a!0) = plr & bd!(a!1) = plr &
>                       bd!(a!2) = plr & bd!(a!3) = plr
>       = hasline plr b bd, otherwise
```

The function `for` doesn't appear to be called.

```
> for :: num->num->num->(num->[char])->[char]
> for a b c f
>       = [], if a > b
>       = f a ++ for (a+c) b c f, otherwise

printboard displays the board in a suitable terminal form

> printboard :: [char]->[char]
> printboard x = pb x 3
>                       where
>                       pb [] n = []
>                       pb (a:b:c:d:e) 0 = a:b:c:d:'\n': pb e 3
>                       pb (a:b:c:d:e) n = a:b:c:d:' ':' ': pb e (n-1)

occupied says whether the square (argument 2) is free in board (argument 1).
```

```
> occupied :: [char]->num->bool
> occupied x n = x!n ~= '.'
```

Reflections

Despite some obvious and radical differences between a functional program to play 3D OXO such as this and the definitive program for playing 3D OXO that is implicit in oxoJoy1994, there are significant points of similarity. The functional programming script (note that this is the term used by functional programmers, even though the script doesn't explicitly prescribe state change) is made up out of definitions of variables of type function (even higher-order functions, that take functions as arguments, and/or return functions as outputs). The way in which this script is incrementally crafted resembles the creation of a definitive script, in that it reflects developing understanding on the part of the programmer. Viewed in this way, the script is a record of states in program development. What is more difficult to sustain is the notion that the variables in the script are associated with observables associated with playing a game of 3D OXO, though indeed they are in a very abstract and sophisticated sense. From a program comprehension perspective, the problem is that these observables - being subjected to highly artificial modes of observation associated with a particular ritualised pattern of interaction - are no longer recognisable as the simple observables they might otherwise be. Consider for example: the current state of the board, the player whose turn it is, or the set of squares that make up a line. What we can't do is to connect these variables with observables in any other ways in which they might participate in a 'non-standard' scenario involving a 3D OXO game, as when a player cheats or a piece falls off the board etc. Note that this poor correspondence between variables and familiar everyday observables seriously undermines the aspiration to human readability. This illustrates a motivating idea in EM: that a formal specification of state, however sophisticated, is not as expressive as an appropriately constructed interactive artefact where the detailed understanding or experiencing of a specific situation is concerned.
