To write a board game language program, first define what the board will look like for the game this is an example:

```
type Player = {A,B}
type Board = Array(3,3) of Player & { Empty }
```

In this case, the (3,3) specifies the dimensions of the board, 3x3.
With the second line, we specify that the board has to be occupied by a player or by empty.
The Player data type has two elements, A and B, so in this case our board has to have either an A, B, or Empty in each tile.
We could expand this further by instead doing something like this,

```
type Board = Array(6,7) of Player & { Empty, Blocked }
```

Now we can have a tile in the board be blocked as well.

After this we define the type of input we will want for our game.
This should be either one value, or a tuple, an example would be:

```
type Input = (Int, Int)
```

Or if we wanted just one number: `type Input = Int`
Later we can use the input keyword to get values from the player.

Now we need to implement the game using functions programmed ourselves.
For Tic Tac Toe game we could write something like this.

```
initialBoard : Board
initialBoard ! (x, y) = Empty

threeInARow : Board -> Bool
threeInARow(b) = or(inARow(3,A,b),inARow(3,B,b))

loop : (Board,Player) -> (Board,Player)
loop(b,p) = while not(gameOver(b,p)) do tryMove(b,p)

isFull : Board -> Bool
isFull(b) = countBoard(Empty,b) == 0

next : Player -> Player
next(p) = if p == A then B else A

-- Game over function !
gameOver : (Board ,Player) -> Bool
gameOver(b,p) = or(threeInARow(b),isFull(b))

isValid : (Board,Position) -> Bool
isValid(b,p) = if (b ! p) == Empty then True else False
```

```
tryMove : PState -> PState
tryMove(b,p) = let pos = input in
                   if isValid(b,pos) then (place(p,b,pos),next(p))
                                     else (b,p)

outcome : (Board,Player) -> Player & {Tie}
outcome(b,p) = if inARow(3,A,b) then A else
               if inARow(3,B,b) then B else
               Tie

play : (Board,Player) -> Player & { Tie }
play(a,b) = outcome(loop(a,b))

result : Player & { Tie }
result = play(initialBoard,goFirst)
```

Combined with the first lines we looked at above, this will make Tic Tac Toe. Here we are using some built in functions as well as one we write ourselves.
When we define a board we start with a name, in this case its initialBoard so we write

```
initialBoard : Board
```

Then we have a series of statements that set the content of the board to start out like:

```
initialBoard ! (x, y) = Empty
initialBoard ! (x, 1) = A
```

This would give us a board that has the first column filled with A's and the rest Empty

The result function serves as running the whole game, calling this will run the game, and the output will be the result of the game. This does a few things, it calls play and passes in the initial board, as well as the player we decided goes first. The play function calls loop, and when it is done will use the outcome function to look at the final board, and decide what the outcome was.

We are using a few builtins here to do work for us, the `InARow` function which checks if there are a certain number of things in a row and returns true if there are. `isFull` looks at the board, and if there are no more Empty tiles it returns true. The `place` function takes in the board, something to place on it, like a player, and a position, and returns a board with this change made. `countBoard` counts the total number of a certain thing, like Empty or a player that are in the board.

We also use the built in function while. This function takes in a function as a condition and a function to execute every loop. If the condition is true it runs the execute function and gives the result as the new input to the condition and execute functions. If the condition is false it returns the last result from the execute function.

The function tryMove uses the input we defined earlier. We say `let pos = input in …`

This will call a builtin function which gets input of the type we defined earlier from the user, in this case it gets two integers. It then sets the variable pos to be the tuple of this integers. Now we can pass this into functions, or return it.

The rest of the code is written using a variety of statements allowed by the language, including if statements, comparison, logical or, and b ! p which returns whatever is in board b at position p.