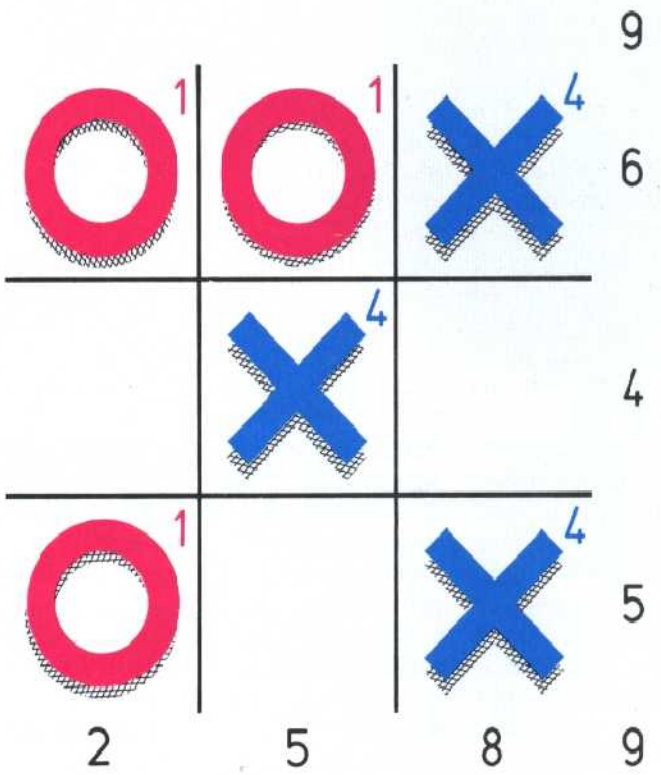
program calculates who won, displaying the result and an accumulating score for itself and its opponent. If the RND function is truly random, then the scores should even out over a large number of rounds, no matter what strategy the player adopts. Now we need to determine how we can improve the computer's strategy to ensure that it will win over a large number of rounds.

When we looked at random functions (see page 209), we learnt that generating a truly random sequence of numbers is an impossible task for both humans and computers, though the latter make a much better approximation. Over many rounds of our game the human player will invariably favour one of the objects more than the others. You can write a subroutine in your program that keeps track of the player's choices, using an array with three elements called, let's say, CHOICE(1), CHOICE(2), and CHOICE(3). Each time the player makes a choice, one is added to the total in the corresponding array element. The computer can then establish which object is more often presented by its opponent, and play the object that beats this preferred choice.

the game. So rather than keep a record of his opponent's choices since the start of the game, it would be better that the program simply recorded, let's say, the last 20 choices. This will require a CHOICE array of 20-by-three elements, and a more sophisticated subroutine to add up the three columns and hence predict the best choice for the computer's next turn.

However, the most serious shortcoming of this algorithm becomes apparent if the player deduces the computer's strategy. Then it is relatively easy for him to play in a way that ensures that the computer will lose on more than half the turns. The player could, for example, consistently play the same object, and then switch to another unexpectedly, and so on. What we need is a different algorithm that avoids these problems. Nevertheless, it would be worthwhile developing programs that use both the fully random and the modified random methods, and observing the scores when these are used by unsuspecting players.

Because humans are incapable of making a totally irrational or random decision, it follows that



Winning Position

'Position evaluation' is fundamental to any board game program — even if the game is as simple as noughts and crosses. In this case, the board is represented as a three by three array, the player's noughts by the value one, and the computer's crosses by a four. Using these values, any position can be evaluated by adding up the totals for every row, column and diagonal. A total of 12 in any of these lines indicates that the computer has won; three means that the player has won; a total of eight shows that two crosses have been played and the computer can win; and so on. The values one and four are used because these ensure that every combination of noughts and crosses gives a unique total

LIZ DIXON

Three problems arise with this method. Firstly, if the computer consistently plays the same object then the player is very quickly going to take advantage of this. Therefore, the computer must generally be made to choose from the three objects using the RND function, while a routine should be added to ensure that it will more frequently choose the object that will beat the player's most preferred choice.

The second problem is that the player will tend to change his favourite object over the course of every choice must be a function of the previous choices. That function may be extremely complicated, and the player almost certainly isn't aware of it, but if the computer can work out a good approximation to that function, then it should be able to win fairly consistently. Because each player will have an individual subconscious formula, and will probably change this formula over the course of a long game, the program must be made to interpret the formula while it is playing. Programs that can learn like this are called