Chapter 8 **Board games and game trees**

In the next two chapters, we are going to look at some of the techniques needed to write programs that play 'board' games. Examples of the sort of game that we have in mind are NIM, Noughts and Crosses (or Tic-Tac-Toe), Kalah, Go-Moku Go, Draughts (Checkers) and Chess. Don't worry if you not familiar with all of these games - we shall explain rules of any games that we use. In all of the above games the outcome depends entirely on the skill of the two players involved. Both players have available exactly the information and their choice of move is not influenced the throw of a dice or the deal of a card. For the time being, we shall avoid consideration of games like Battleships and Cruisers where neither player can see other's board or Backgammon where the throw of a dice affects the choice of moves available.

In Chapter 1, we presented the outline program structure needed for a program that plays a board game. In the next two chapters, we will look at techniques that can be used by such a program for choosing a goal move.

In this chapter, we shall concentrate on rather trivial games. It may seen that these games are a long way removed from more intellectually demanding games like chess and draughts, but the techniques that we introduce by using these simple games are easily modified to deal with difficult games. The modifications needed are discussed the next chapter.

There are three main problem areas in programming games of the type that we are considering:

(a) Choosing a move:
    In the next two chapters we shall devote a great deal of attention to the techniques that are available for writing procedures to choose goal moves.

(b) Representing board positions and moves inside the computer:
    For a given game, there may be several different ways of representing a board position inside a program. In our 'Last One Wins' program, a single integer was enough to represent the 'board'. In more difficult games, there may is a choice of data structure representations tor a board position. Different

representations for the Noughts and Crosses board suggested at the end of Chapter 1.

(c) Displayipg the board:
Producing elegant displays of board positions on the screen involves the use of graphics facilities that are extensively discussed elsewhere.

8.1 Game trees

Choosing a sensible move in a board game often involves exploring ahead from the current position in the game and considering the various sequences of moves and counter-moves that are available from the position. It will make our discussion of this process easier, if we introduce the idea of a 'game tree'. To do this, we use a variation of the game 'Last One Wins' (Chapter 1). In order to better illustrate a number of points we change the rules of the game as follows:

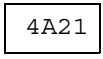
> Players now score a point for each counter removed during the course of a game. The player who makes the last move scores an additional two points. The aim of the game is to maximise the 'point difference' between oneself and one's opponent.

We shall call this version of the game 'Last One Wins - or does he?'.

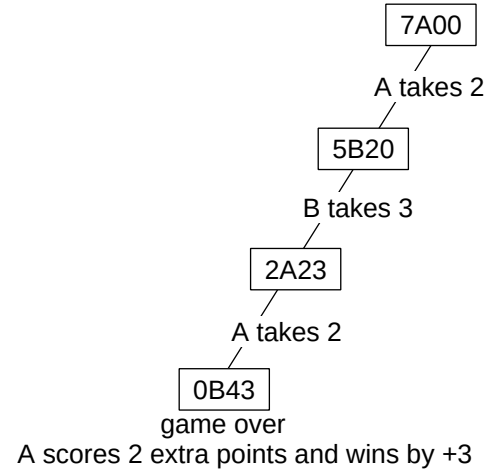
There are a number pieces of information required to completely describe a position reached during the course of a game:

> The number of counters left on the board;
> Whose turn it is (player A or player B);
> A's score so far;
> B's score so far.

We shall represent a state of the game graphically by a box containing this information. For example:



represents a state of the game in which there are 4 counters left on the board, it is A's turn to play, A has scored 2 points so far and B has scored 1 point so far. A move can be represented by a line joining two such boxes and the sequence of moves played in a particular game starting with 7 counters (A starts) might be:



There is, of course, usually a choice of moves available in any one position and the complete set of possibilities available for the game with 5 counters (A starts) can be illustrated by the tree on the next page.

Any path starting at the 'root' of the tree (at the top) and moving down through the tree to a 'leaf' represents a particular sequence of moves making up one complete game that could be played. In this tree, there are represented 13 possible games. In terms of the tree, this means that there are 13 different leaves and 13 different pathways from root to leaf. The terminal positions or leaves in the tree are marked with the point difference for A and these values will be used later.

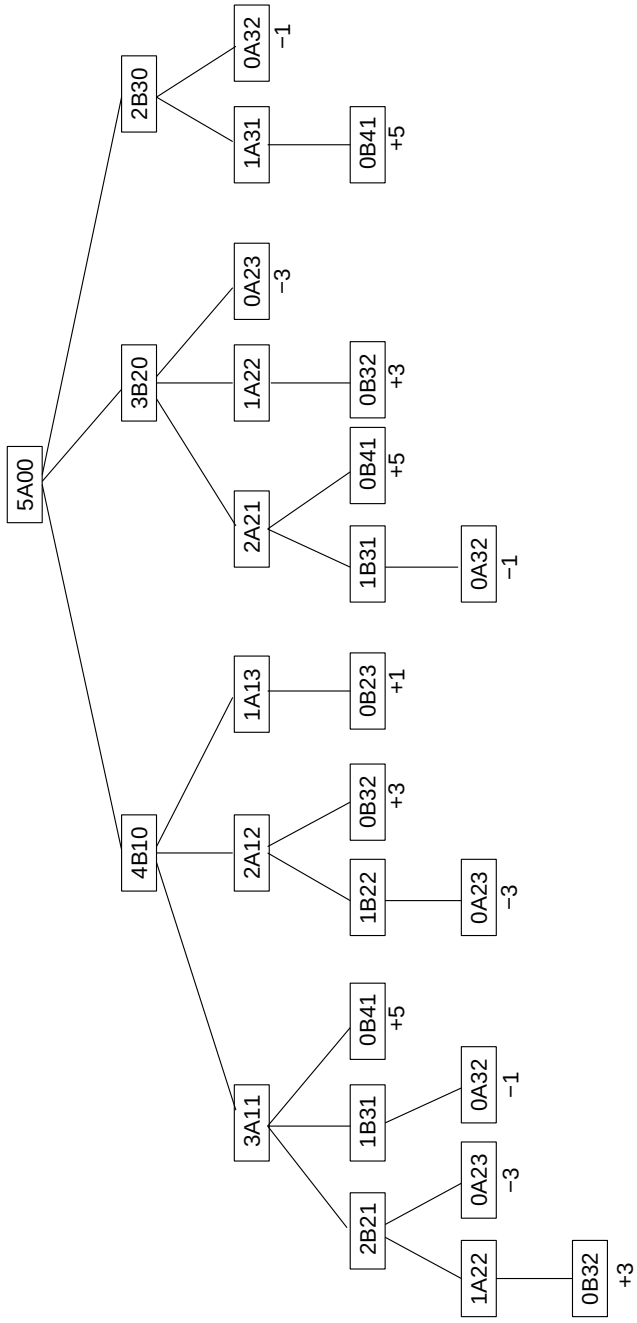
The boxes representing positions are often referred to as the 'nodes' of the game tree.

Exercises

1 The rules for 'Grundy's Game' are:

> Two players start with a pile of counters on the board between them. The first player divides the pile into two unequal piles. The players alternately do the same to one of the remaining piles. The player who first cannot play loses. (This happens when all the piles on the board contain 1 or 2 counters - a pile of two counters cannot be subdivided into two unequal piles.)

Draw a game tree for Grundy's Game starting with a pile of 7 counters.


5A00
4B10
3B20
2B30
3A11
2A12
1A13
2A21
1A22
1B31
0A23
1B22
0A32
2B21
0B41
0B32
0B23
+5
+3
+1
−3
−1

8.2 Using recursion to generate a game tree

Our purpose in introducing the idea of a game tree is to enable us to write a procedure that decides, for a given position, which particular move is best from the point of view of the player whose turn it is. In order to use the game tree to help in its choice of moves, such a procedure will have to explore the various branches of the game tree. Before introducing the complication of comparing different sequences of possible moves, it will be instructive to write a procedure that generates and prints the entire game tree for 'Last One Wins - or does he?'. In fact we shall see later that this procedure can be fairly easily modified to collect the information needed to select a good move on the basis of its exploration of the tree.

We can outline what our procedure must do:

```
To generate the game tree from position P:

Print position P.
Work out which moves are available in P (if any).

FOR each move available
   Construct the position we get if we make that move.
   Generate the game tree from this new position.
NEXT move
```

The way that we have described this process immediately suggests the use of recursion (Chapter 7). In this case, generating a tree is described in terms of generating some smaller trees. The word 'smaller' is important - it is this aspect of our definition that makes sure that the recursive description eventually stops when a position is found in which no moves are available. Here is a recursive procedure for exploring the game tree.

```
100  DEF PROCgrowtree(counters, turn$, Ascore, Bscore)
110  LOCAL move,movesavailable,
       newturn$,newAscore,newBscore
120  PRINT ;counters; turn$; Ascore; Bscore
130  IF counters = 0 THEN
       PROCfinalscore : ENDPROC
140  PROCcheckmovesavailable :REM defined as before.
150  FOR move= 1 TO movesavailable
160    IF turn$="A" THEN newturn$ = "B" :
         newAscore=Ascore+move: newBscore=Bscore
       ELSE newturn$ = "A" :
         newBscore=Bscore-move: newAscore=Ascore
170  PROCgrowtree(counters-move, newturn$,
       newAscore, newBscore)
180  NEXT move
190  ENDPROC
```

```
200  DEF PROCfinalscore
210    PRINT "Final Score: ";
220    IF turn$="A" THEN PRINT ;Ascore; " - "; Bscore+2
         ELSE PRINT ;Ascore+2; " — "; Bscore
230  ENDPROC
```

The procedure takes some parameters that represent the position frcan which it is to generate the game tree. Note that the definition of PROCgrowtree reflects the outline and contains a call of itself.

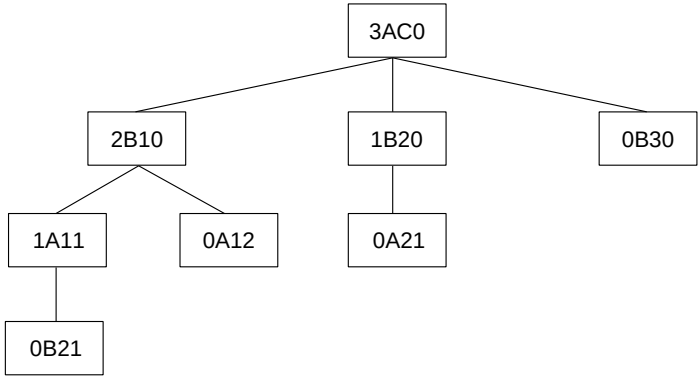
In order to generate the tree for the game with 3 counters, A starts, we need to insert:

```
10  PROCgrowtree(3, "A", 0, 0)
20  END
```

A run of this program produced the following output:

```
3A00
2B10
1A11
0B21
Final score: 4 - 1
0A12
Final score: 1 - 4
1B20
0A21
Final score: 2 - 3
0B30
Final score: 5 - 0
```

The game tree for the game with 3 counters is:

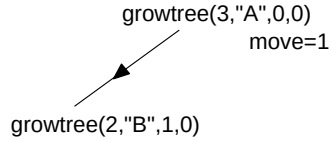


As we can see, our recursive procedure has generated and

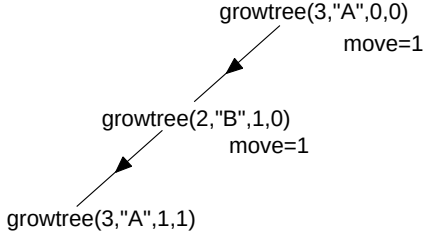
printed all the nodes in the tree. We shall adjust the program! so as to improve the layout of its output in a moment, but before doing this it will be instructive to look in some detail at how it works. The program starts by calling:

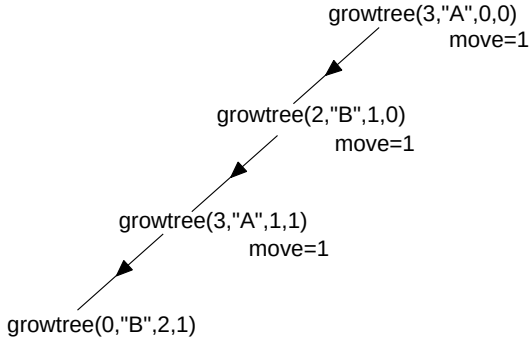
```
PROCgrowtree(3,"A",0,0)
```

We can think of this procedure call, together with its parameters, as representing the root of the game tree. In the course of the evaluation of this procedure call it sets move=1 and calls itself recursively (at line 170). We can represent the situation at this stage by:



Because this new procedure call is entered before the previous one is exited, we now have two calls of the same procedure active at once, each with its own private set of parameters and each with its own private copies of its LOCAL variables. As we suggested in Chapter 7, you may find it easier to think in terms of two completely separate copies of the procedure, although of course it does not work like this behind the scenes. When the computer eventually exits from the second procedure call, it will carry on obeying the first where it left off, but that will not happen until the 'subtree' to be grown by the second procedure call has been completely generated. Having printed the values of its parameters, the second procedure call sets its local variable move=1 and calls PROCgrowtree recursively (at line 170). We can now picture the situation as:





This procedure activation makes one more recursive call and on entry to this last call of the procedure the IF statement at line 130 is triggered. This outputs the final score for the current position and terminates the procedure call. The computer exits to the prevoious procedure call

```
growtree(1,"A",1,1)
```

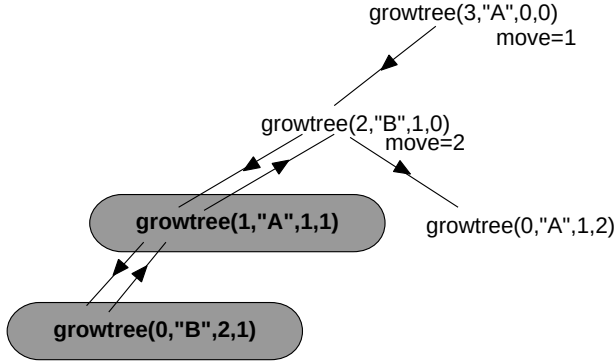
and carries on from where it left off there (line 180). There are no more moves available in the position represented by that procedure activation and so the FOR loop terminates and that procedure activation is exited. The computer is now continuing its execution of the call

```
growtree(2,"B",1,0)
```

and again carries on at line 180. In this case, the FOR loop is executed for a second time with move=2 and this results in the procedure call

```
growtree(0,"A",1,2)
```

The new situation is illustrated in the following diagram:



In order to present a complete picture of what has happened as well as what is happening, the diagram includes the procedure calls that have been terminated, but these are clearly marked.

As you can see, the program is working its way systematically through the nodes in the tree by proceeding as far as it can down one branch before retracing its steps and exploring another branch. This type of exploration is known 'depth-first search' and the process of going back and trying another branch is known as 'backtracking'. You will observe a similar effect if you try to draw a tree without lifting the pen off the paper and without retracing your steps more than is absolutely necessary.

The latest procedure activation represents another position in which the end of the game has been reached and after printing the final score for this position, the procedure call is terminated. The program retraces its steps to line 180 in the call

```
growtree(2,"B",1,0)
```

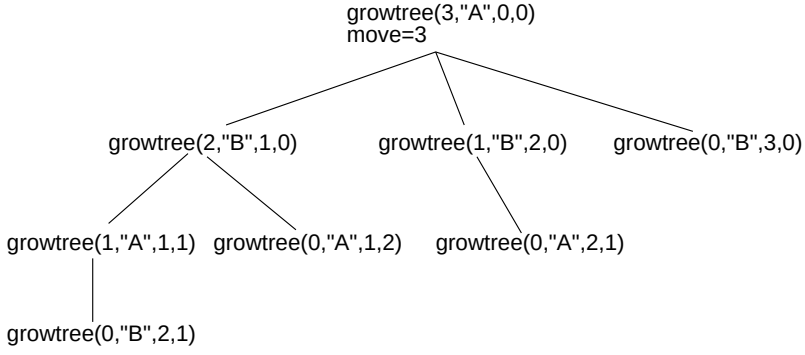
which also terminates as there are no more moves available ta this position. This takes us back to line 180 in the procedure activation that started the whole process off. The EOR loop here is repeated with move=2 and this results in the call

```
growtree(1,"B",2,0)
```

which in turn calls

```
growtree(0,"A",2,1)
```

Then the final score for this position has been printed, the program retraces its steps back through the last two procedure calls to the topmost procedure call in which the FOR loop is executed with move=3. This results in the situation illustrated in the next tree.



A final score is output for the last time and the two remaining procedure calls exit in turn.

You should now be able to see how tho program keeps track of the structure of tho game as it is being generated. If recursion were not available, the game positions generated and information about which moves have been tried in each position would have to be stored in some sort of data structure that would allow the program to retrace its steps when all the possibilities down one branch of the tree have been considered. The data structure that would be needed to do this is called a 'stack'. In a programming language that allows recursion, we do not need to create such a data structure — we can use the procedure entry and exit mechanisms to handle the 'bookkeeping' details needed to implement the backtracking process. (In fact the computer uses its own stack behind the scenes to keep track of recursive procedure calls.) When we use recursion, trying a move corresponds to calling a procedure - the resultant procedure activation and its parameters represent the new game position. Note the importance of the LOCAL declaration at the head of the procedure. Several procedure activations can be in existence at once, and each one needs its own private copy of information that is unique to that activation and the position that it represents.

One final point: if you try to use PROCgrowtree for more than 10 counters, it will fail with the error message 'Too many FORs'. There is a limit to the number of FOR's that can be 'nested' inside one another and when we call our procedure recursively it behaves as if the FOR loop obeyed in the recursive call were inside the FOR loop of the procedure that made the call. (This problem was discussed in Chapter 7.) There are over 1100 positions in the tree for the ganre starting with 11 counters, so you are unlikely to want to print them out. If you do want to use the procedure for 11 or more counters, you will have to replace the FOR-NEXT with an IF-GOTO loop or a REPEAT loop (REPEAT loops can be more deeply nested than FOR loops).

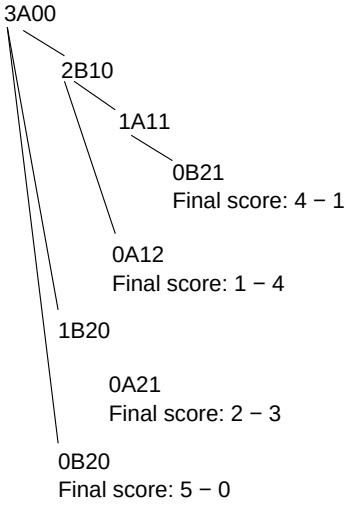
Now that we have seen how our tree generating program works, it is interesting to see if we can improve the layout of the output produced. One way of doing this is to make the output reflect the structure of the tree being explored by printing some extra spaces at the start of each line of output. The number of spaces printed before a position is proportional to the depth of the position in the game tree. The easiest way of doing this is to add an extra parameter to our recursive procedure. Remember that a call of the procedure represents a game position. Each time we call the procedure we shall supply an extra parameter that indicates the depth of the new position in the game tree. We need to make the following changes to our procedure:

```
100  DEF PROCgrowtree(counters, turn$,
       Ascore, Bscore, depth)

110  LOCAL move,movesavailable,
       newturn$,newAscore,newBscore
115  PRINT
120  PRINT TAB(4*depth) ;counters;turn$;Ascore;Bscore
130  IF counters = 0 THEN PROCfinalscore : ENDPROC
       :
       :
170  PR0Cgrowtree(counters-move, newturn$,
     newAscore, newBscore, depth+1)
       :
       :
200  DEF PROCfinalscore
210  PRINT TAB(4*depth); "Final score: ";
       :
       :
```

Note that when an activation of the procedure tries a move by calling the procedure recursively, the depth of the new position created is one more than the current depth. Hence the need to supply depth+1 as a parameter to the recursive procedure call at line 170. When the procedure call corresponding to depth=depth+1 is exited, The old value of depth is restored. If the procedure is now called by

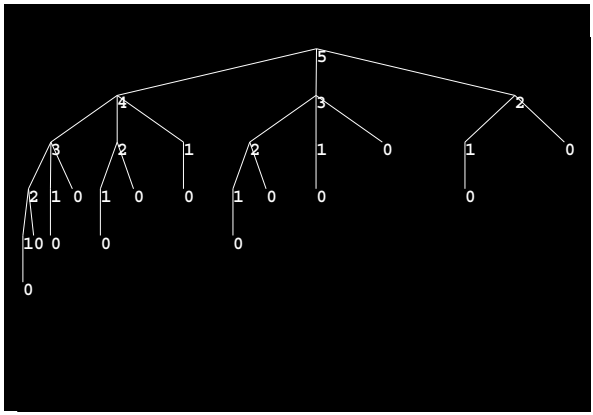
```
  10  PROCgrowtree(3, "A", 0, 0, 0)
```

the output produced is



The lines have been drawn to make it clear how the layout of the program output corresponds to the structure of the game

tree. A rather interesting variation of the above program would be one that uses graphics statements to draw a complete game tree.



Here is the program that produced the photograph:

```
 10  INPUT "Counters", counters
 20  MODE 4
 30  MOVE 640,1000
 40  VDU 5
 50  PROCdrawtree(counters, 640,1000,1280)
 GO  k=GET:MODE 7
 70  END

100  DEF PROCdrawtree(counters, x,y, width)
110  LOCAL moveanovesavailable, newx, newy, newwidth
120    PRINT ;counters
130    IF counters=0 THEN ENDPROC
140    PROCcheckmovesavailable
150    newwidth = INT(width/movesavailab1e)
160    newx = INT(x-width/2+newwidth/2)
170    newy = y-100
180    FOR move= 1 TO movesavailable
190      MOVE x,y : DRAW newx,newy
200      PROCdrawtree(counters-move,newx,newy,newwidth)
210      newx = newx + newwidth
220    NEXT move
230   ENDPROC

 250  DEF PROCcheckmovesavailable
 260   IF counters<3 THENmovesavailable = counters
       ELSE movesavailable = 3
 270  ENDPROC
```

Exercises:

1 Draw the game tree of procedure calls that takes place as a result of obeying the statement

```
PROCgrowtree(4, "A" ,0 ,0)
```

Make sure that you understand the order in which procedure activations are entered and exited.

8.3 Manipulating board positions during recursion

In the procedures written in the last section, a board position could be easily represented by a single integer. When PROCgrowtree was called recursively, it was given a value for a parameter ('counters') representing a new board position. When the recursive procedure call exits, that value for 'counters' disappears and the old value is now available for trying other moves.

More complicated data structures such as arrays cannot be passed as parameters in BBC BASIC and we need to find some other way of restoring a board position to the state it was in before a move was tried by a recursive call of the procedure. One way of doing this is to make the move by changing the board before calling the procedure recursively and then 'unmake' the move after the recursive procedure call has exited. This is illustrated in the following program that prints the tree for Grundy's Game.

If you wonder why we have used a REPEAT at line 350 instead of a FOR, try replacing the REPEAT with a FOR and refer back to page 267.

```
 10  DIM pile(20)
 20  pilet(1) = 7 :REM start with one pile
 30  PROCgrowtree("A", I) :REM of 7 counters.
 40  END

100  DEFPROCgrowtree(turn$, piles)
110  LOCAL p, pilestried, nextturn$
120    PRINT turn$;
130    FOR p=1 TO piles : PRINT ;pile(p); : NEXT
140    PRINT
150    IF turn$="A" THEN nextturn$="B"
         ELSE nextturn$=" A"
160    pilestried = 0
170    FOR p = 1 TO piles
180      IF pile(p)>2 THEN PROCtrysplits
190    NEXT p
200    IF pilestried=0 THEN
         PROCfinalresult :REM there were no moves.
210  ENDPROC
```

```
300  DEF PROCtrysplits
310  LOCAL posssplits, split
320    pilestried = pilestried+1
330    posssplits = INT((pile(p)-1)/2)
340    split = 0
350    REPEAT
360      split = split+1
370    pile(piles+1) = split
380    pile(p) = pile(p) - split
390    PROCgrowtree(nextturn$, piles+1)
400    pile(p) = pile(p) + split :REM 'unmake' mve.
410  UNTIL split=posssplits
420  ENDPROC

500  DEF PROCfinalresult
600    IF turn$ = "B" THEN PRINT " A wins."
         ELSE PRINT " B wins. "
700  ENDPROC
```

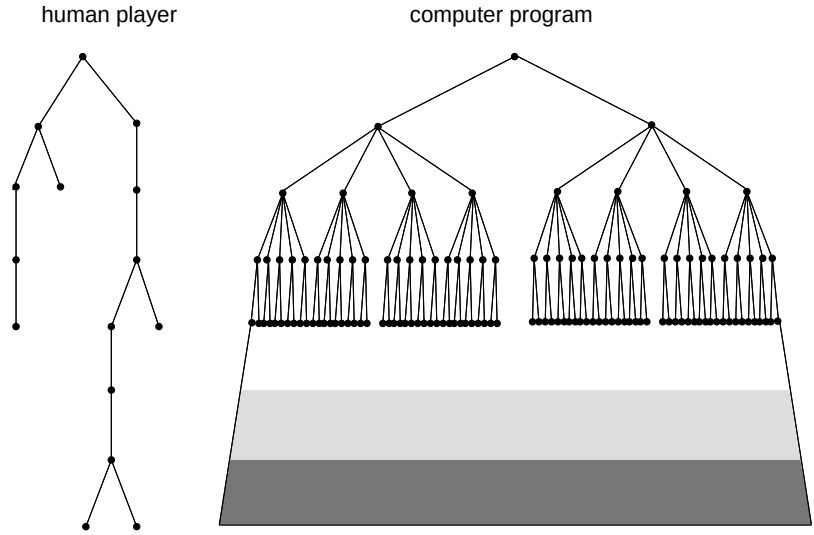
The program also exhibits some other interesting features that you should study. For example, the recursion is no longer quite so obvious - PROCgrowtree uses a subsidiary procedure, PROCtrysplits, to try different ways of splitting a given pile and it is this procedure that may call PROCgrowtree again. Also there is no explicit test at the head of PROCgrowtree to terminate the recursion - recursion terminates if there are no piles to be subdivided and this is not known until all the piles have been examined.

Exercises

1 Draw the tree of procedure calls that take place as a result of obeying the above program. Mark alongside each procedure call the state of the array 'pile' when the procedure is entered.

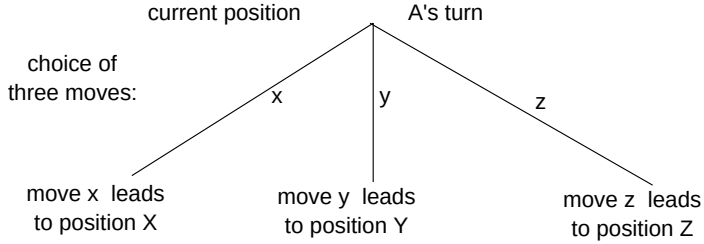
8.4 Minimaxing

We now need to introduce a criterion that can be used by a game playing program to decide on the best move in a given position. When a good human game player is presented with a board position in which he has to make a move, he usually considers some of the moves available, some of his opponents possible responses to these moves, some of his possible responses to his opponent's moves, and so on. In other words, he looks ahead and explores part of the game tree rooted at the current position. Human game players are very selective about which branches of the tree they explore - they have to be because of their slow processing speed and limited short term memory capabilities. A typical human player thinks about only one or two moves at each position in his lookahead, whereas a computer program usually explores a much bushier game tree:



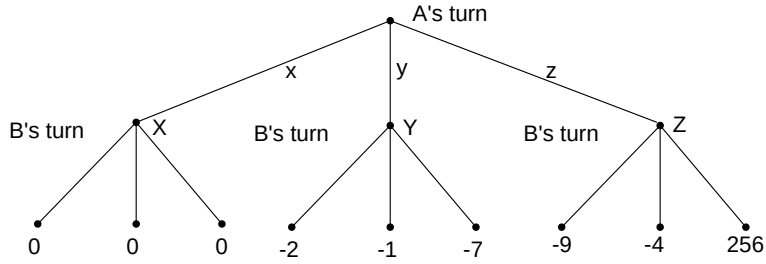
The better the hunman player, the better he is at recognising the promising branches of the tree that need exploration. This process of ignoring branches of the lookahead tree - 'tree pruning' - is very important for programs too, and will be discussed later, but for the time being we shall assume that a game playing program will explore the entire game tree in deciding on its move. Such a complete exploration is possible only for trivial games like 'Last One Wins - or does he?', but the programming techniques required for the partial exploration of a much larger tree are almost identical and will be discussed later.

Consider the problem of deciding on a move in the following situation.



One way of making a choice of moves would be to give

numerical values to position X, Y and Z where the value assigned to a position indicates how good that position is from player A's point of view. A could then choose the position with the highest value. It is easy to assign such a value to a terminal position of a game tree. In a position where the game is over, we can compute a final score for the game. For example, in the tree for 'Last one Wins - or does he' each terminal position was given a value representing the point difference by which player A has won. A negative value means that player A has lost by that amount. In a game where the outcome must be just win, draw or lose, we could assign values +1, 0 or -1 to the terminal nodes of the tree. Thus if X, Y and Z were all terminal positions, we could easily give values to these positions for the purpose of comparing moves x, y and z. A problem arises when X, Y and Z are not terminal positions and player B now has a choice of moves in each of these positions. Let us extend the above tree to one more level. We assume that there are three moves available in each of positions X, Y and Z and that any one of these moves will terminate the game. We have assigned hypothetical final scores to each of the terminal positions.



We assume that positive values represent a win for A (the higher the better) and negative values mean that A has lost. It is highly unlikely that an actual game would have such widely differing terminal values but we have chosen values that will emphasise the points that we wish to make. We must now decide how to evaluate the non-terminal positions X, Y and Z.

All three moves available in position X lead to scores of 0. Whatever move player B chooses in position X must inevitably lead to a draw, and there is no difficulty in deciding that position X should be given a value of 0.

The three moves available in position Y all lead to positions where A loses by varying amounts. It is clear that we can classify position Y as a losing position and assign it a negative value. We shall decide exactly what negative value it should be given when we have considered position Z.

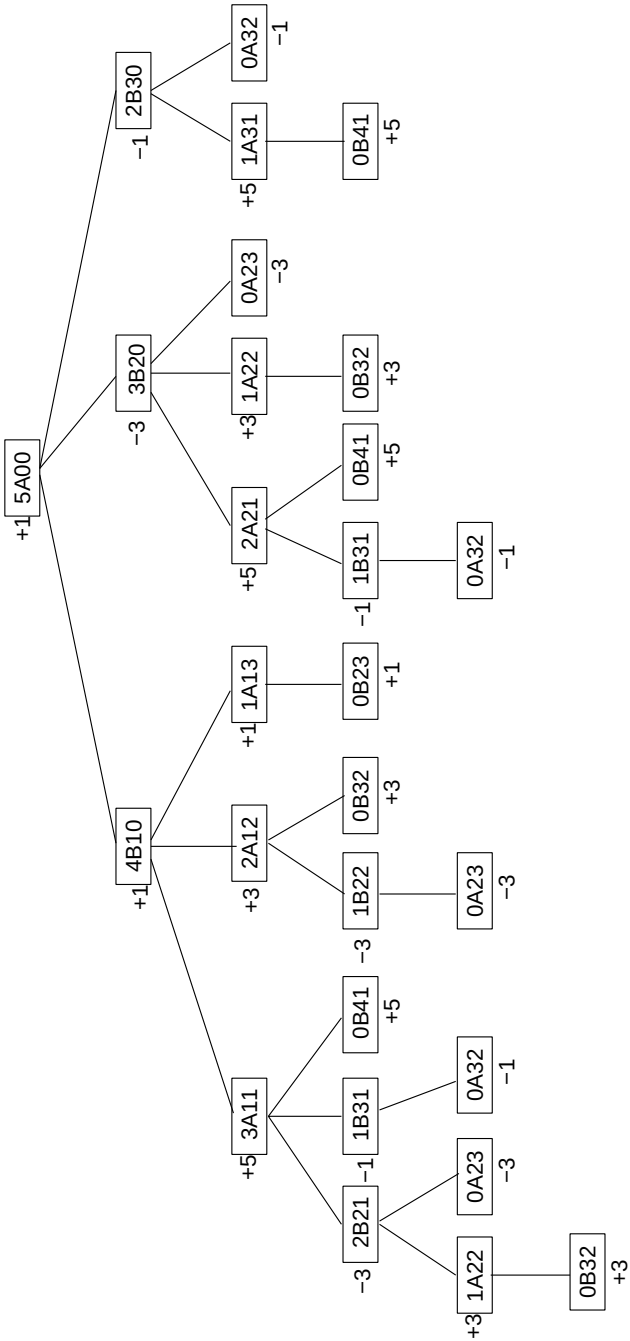
In position Z, B has a choice of moves, one of which leads to a spectacular win for playor A. In evaluating position Z, player A must decide what I likelihood there is of achieving this spectacular win. After player A baa made a move control of the outcome will pass to player B and, in order to decide on the value of position Z, A must make some assumption about how player B will choose his move. If A assumes that B is making completely random moves, then he might decide that the value of position Z is the average of the values of the three positions that B might move to. This would be a reasonable assumption in a game where B's move is determined by the throw of a dice, but in games of the type that we are dealing with both players have exactly the same information available and both players have a completely free choice of move. Under these circumstances, experienced board game players know that the only safe assumption is that one's opponent will choose his best move in any position. In our imaginary game, this means that player A assumes that it player B is presented with position Z then B will make his best move, ie. the move leading to the lowest possible value from A's point of view. Under this assumption, the value of Z is the minimum of the values of the three positions to which moves could be made. We therefore say that Z has the value -9. (In the same way, the value of position Y would be -7.)

We can now see that this part-tree represents a situation in which A can only win by a fluke. The correct move by player A must be to make the move leading to the highest valued position, ie. move x. When player A is analysing the game tree and values are being given to positions from his point of view, the above assumption together with the fact that A will always choose his best move gives us the following two rules:

(a) The value of position at which it is A's turn is the maximum of the values of positions to which moves are available.

(b) The value of a position at which it is B's turn is the minimum of the values of positions to which moves are available.

You should now examine the game tree for 'Last one Wins - or does he' and attempt to apply these rules in order to give values to the non-terminal nodes. You should find that you an do this provided that you work backwards from the terminal nodes. For obvious reasons this process is known as minimaxing'. In the next tree we have done this and all the non-terminal nodes have been given values, including the node at the root of the tree.


5A00
+1
4B10
+1
3B20
−3
2B30
−1
3A11
+5
2B21
−3
1B31
−1
0B41
+5
0A32
−1
0A23
−3
1A22
+3
0B32
+3
2A12
+3
1B22
−3
0B32
+3
0A23
−3
1A13
+1
0B23
+1
2A21
+5
1B31
−1
0B41
+5
0A32
−1
1A22
+3
0B32
+3
0A23
−3
1A31
+5
0B41
+5
0A32
−1

The value that has been assigned to the position at which the game starts tells us that if both players always make their best moves, then the outcome of the game is bound to be score of +1 tor the-player who starts (A in this case). If player B ever chooses a move that does not lead to the position with mininium value, then A will improve on this score. You should convince yourself of this by trying different sequences of moves in the tree.

The fact that the minimax process determines a unique value for the root node of the entire game tree is sometimes referred to as the 'Foregone conclusion Theorem'. In theory, the outcome of any game of the type that we are considering is a foregone conclusion. However, the trees for all but the most trivial of games are extremely large. For example, on the next page we have a fragment of the game tree for noughts and crosses.

Even with a simple game like this, the entire game tree could easily be a mile or more across. However, the noughts and crosses board is highly symmetrical and the size of the tree could be considerably reduced by eliminating duplicate positions that are reached by different sequences of moves and by eliminating positions that are rotations and-reflections of other positions (although it is not easy for a program to do this). The foregone conclusion of noughts and crosses is well known to be a draw.

If we move to more serious games like draughts and chess, the problem of determining the 'foregone conclusion' becomes many orders of magnitude worse. It has been estimated that

The draughts tree contains about 10^{40} nodes.

The chess tree contains about 10^{120} nodes.

If these numbers mean very little to you, the following facts may help to put them in perspective.

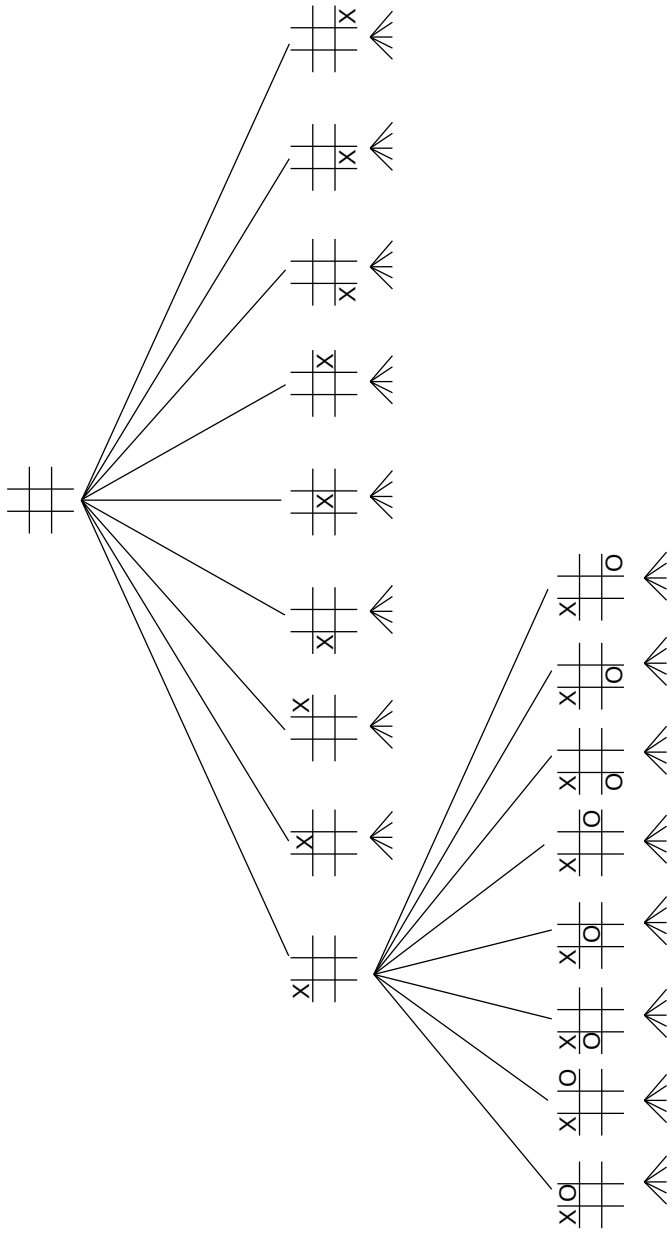
There are only 10^{16} microseconds in a century and it takes about 1 microsecond for a computer to carry out a simple operation such as the addition of two numbers.

Cosmlogists estimate that there are about 10^{80} elementary particles in the universe.

Thus it is exceedingly unlikely that we shall ever know what the foregone conclusion is for chess or draughts, even with the help of high-speed computers.

Exercises

1 Determine the foregone conclusion for the version of Grundy's Game that starts with 7 counters. (A win for A = +1, a win for B = -1.)



8.5 A recursive function for minimaxing

Despite the remarks made at the end of the last section, the process of minimaxing still plays an important role in programs that play more difficult games like draughts and chess. Before modifying our analysis to deal with such games, we shall use the game 'Last One Wins - or does he?' to introduce the programming techniques required to carry out minimaxing. We are going to modify the procedure 'growtree' of Section 8.2 so that it explores the entire game tree as before, and in the process calculates minimax values for the nodes. The nodes of the tree will no longer be printed as they are generated.

Once you have gained some confidence in handling recursion you will find that you can write a recursive procedure or function and be sure that it is correct without subsequently going through a detailed analysis of how it works. You must acquire the ability to write a recursive description of the process you are programming without attempting to visualise in detail how the procedure or function will behave when the program is running. In the present context, we can outline the process of calculating a value for a node as follows:

```
To calculate a value for a given position:
   IF the position is a terminal position THEN
      calculate the final score
   ELSE
      FOR each move available
         Calculate value for position reached by move
         Keep a note of the best (max or min) value found
      NEXT move
      Best value found is the value required.
```

The recursive nature of this description should be obvious, Provided that we have correctly described the process of calculating a value for a given position in terms of the values of the positions to which moves are available, then the recursion mechanism will automatically apply our correct description at all levels in the tree. The program will search recursively down the tree until it reaches terminal nodes from which values can be carried back up the tree.

Note the similarity in structure between this description at the minimax process and the procedure 'growtree' programmed in Section 8.2. In this case, it is convenient to define a function rather than a procedure, the value produced by a function call being the minimax value for a node. Conversion of' the above outline description into a recursive Basic function is fairly straightforward:

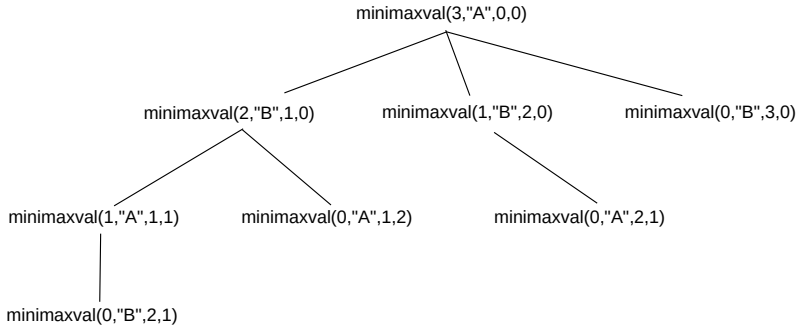
```
100  DEF FNminimaxval(counters, turn$, Ascore, Becore)
110  LOCAL move, movesavailable, neturn$,
         newAscore, newBscore, bestsofar, nextval
120    IF counters = 0 THEN = FNfinalscore
130    PROCcheckmovesavailable
140    IF turn$="A" THEN bestsofar = -100
         ELSE bestsofar = +100
150    FOR move = 1 TOmovesavailable
160      IF turn$=" A" THEN newturn$ = "B" :
           newAscore=Ascore+move: newBscore=Bscore
         ELSE newturn$ = "A" :
           newBscore=Bscore+move: newAscore=Ascore
170      nextval = FNminimaxval(counters-move,
           newturn$, newAscore, newBscore)
180      IF turn$="A" THEN
           bestsofar=FNmax(bestsofar,nextval)
         ELSE bestsofar=FNmin(bestsofar,nextval)
190    NEXT move
200  = bestsofar

300  DEF FNfinalscore
310    IF turn$="A" THEN = Ascore-(Bscore+2)
         ELSE = (Ascore+2)-Bscore

400  DEF FNmax(old,new)
410    IF new<o1d THEN =old
         ELSE =new

420  DEF FNmin(old,new)
430    IF new>old THEN =old
         ELSE =new
```

As was the case with the procedure 'growtree', an activation of the above function represents a node in the game tree. The nodes are examined in exactly the sane order as they were by 'growtree'. Instead of printing the node represented by the paranieters of a function activation, the minimax value calculated by a call of the function is returned as the result of that function call. The value returned by a recursive call of the function is immediately examined by the function activation that made the call (line 180).

In order to discover the 'foregone conclusion' of the game, with 3 counters (A starts), we could call our function ,as follows:

```
10  PRINT "3 counters - foregone conclusion:" ;
20  PRINT ; FNminimaxval(3, "A", 0, 0)
30  END
```

This prints the minimax value of the initial position in the game starting with 3 counters. The tree of function calls produced during a run of this program is:



This tree of function calls is generated in the same way as the tree of procedure calls discussed in detail in Section 8.2. You should annotate each function call in the tree with the minimax value that is the eventual result of that call.

8.6 Mutually recursive functions for mimaxing

The purpose of a call of PROCgrowtree was simply to print a representation of a node of the game tree. The operations carried out by FNminimaxval depend to a much greater extent on the question of whose turn it is. For this reason, the above function looks very cumbersome because of the repeated need to examine the value of the parameter 'turn$' in order to decide whether the current function activation represents a position at which it is A's turn (maximising) or B's turn (minimising). We can produce a considerably more elegant program for minimaxing if we define two mutually recursive functions. One function, FNmaxval, will be use to calculate the minimax value of a node at which it is player A's turn md the other, FNminval, will be used to calculate the minimax value of a node at which it is player B's turn. FNmaxval will try each move available in a position and must use FNminval to evaluate the position reached by each move. In a similar way, FNminval uses Flimaxval.

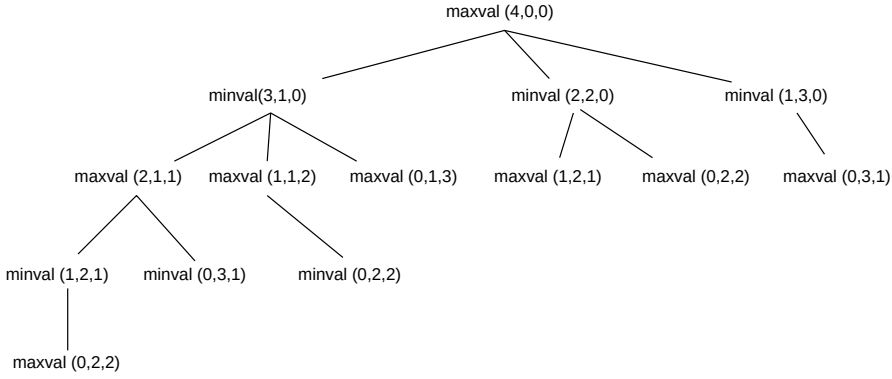
```
100  DEF FNmaxval(counters, Ascore, Becore)
120  LOCAL move, movesavailable, maxsofar, nextval
130    IF counters = 0 THEN = Ascore-(Bscore+2)
140    PROCcheckmovesavailable
150    maxsofar = -100
160    FOR move = 1 TO movesavailable
170      nextval = FNminval(counters-move,
           Ascore+move, Bscore)
180      maxsofar=FNmax(maxsofar ,nextval)
190    NEXT move
200  = maxsofar
```

```
300   DEF FNminval(counters, Ascore, Bscore)
320   LOCAL move, movesavailable, minsofar , nextval
330     IF counters = 0 THEN = (Ascore+2)-Bscore
340     PROCcheckmovesavailable
350     minsofar = +100
360     FOR move = 1 TO movesavailable
370       nextval = FNmaxval(counters-move,
            Ascore, Bscore+move)
380       minsofar=FNmin(minsofar,nextval)
390     NEXT move
400   = minsofar

500   DEF FNmax(old,new)
510     IF new<old THEN =old
          ELSE =new

520   DEF FNmin(old,new)
530     IF new>old THEN =old
          ELSE =new
```

Note that the parameter 'turn$' is no longer needed. To print the 'foregone conclusion' of the game starting with 4 counters we now need:

```
10   PRINT "4 counters - foregone conclusion:";
20   PRINT ; FNmaxval(4, 0, 0)
30   END
```

The tree of function calls resulting from a run of this program is:



The nodes of the game tree are examined in the same order as before, but activations of FNmaxval represent positions at which it is A's turn and activations of FNminval represent postions at which it is B's a turn.

We have mentioned before that our emphasis in the book is on clarity of presentation of ideas. There are various ways in which the above program could be changed so as to reduce the amount of work that it has to do. For example, we could replace the two parameters 'Ascore' and 'Bscore' by a single parameter 'Asnetscore' whose value is the difference between A's score and B's score. Other more subtle improvements could be made, but only at the cost of obscuring the way that the function works.

Finally, note that none of the above minimaxing functions, as presented, will work for more than ten counters (see page 267 for a discussion of the reasons for this).

Exercises

1 Write functions FNminval and FNmaxval for calculating the minimax values of positions in Grundy's Game.

2 If you are familiar with any other 'simple' board games, such as NIM, do the same for them.

8.7 Choosing a move in a 'small' game.

Before studying the problem of choosing a move in a more difficult game, we shall finish our study of 'small' games by examining a number of ways in which a program can be made to choose its move in such a game. The program at the end of Chapter 1 is easily modified to play 'Last One Wins - or does he?'. We need two variables 'Apoints' and 'Bpoints' initialised to zero at the start of PROCaygame:

```
105  Apoints = 0 : Bpoints = 0
```

Making a move now involves adjusting the appropriate player's score:

```
750  counters=counters-move : Apoints=Apoints+move
     :
920  counters=counters-move : Bpoints=Bpoints+move
```

and the program must now announce the final score at the end

of the game:

```
610  IF turn$="A" THEN Bpoints = Bpoints+2
       ELSE Apoints = Apoints+2
611  PRINT "Final Score -   ME:"; Apoints
612  PRINT "               YOU:"; Bpoints
```

The techniques introduced in this section can now be described by reprogramming PROCplayerA in various ways.

Exhaustive lookahead with minimaxing
When presented with a position, the program can use the minimax functions defined in Section 8.6 to carry out an exhaustive lookahead along all possible sequences of moves from the current position. The result of this lookahead will be to obtain a value for each of the positions that can be reached by a legal move from the current position. The program should then choose the move leading to the position with the highest value. The following version of PROCplayerA does this.

```
700  DEF PROCplayerA
710  LOCAL move
720    move = FNbestmove
730    PRINT "I take "; move; " counters."
740    counters=counters-move : Apoints=Apoints+move
750    turn$ = "B"
760  ENDPROC

1100 DEF FNbestmove
1110 LOCAL move,movesavailable,nextval,max,bestmovesofar
1120   PROCcheckmovesavailable
1130   max = -10
1140   FOR move = 1 TO movesavailable
1150     nextval = FNminval(counters-move,
           Apoints-tmove, Bpoints)
1160     IF nextval>max THEN
           max = nextval : bestmovesofar = move
1170   NEXT move
1180 = bestmovesofar
```

FNmaxval and FNminval will have to be renumbered from 1200 onwards. You should notice the similarity between FNbestmove and FNmaxval. The difference is that FNmaxval is used to find the value of the best position to which moves are available, whereas in the above context we wish to find the move that leads to the best position.

Decision tables

In small games like Noughts and Crosses or 'Last One Wins', the number of different positions that might be encountered by a program is small enough for a program to store a table containing a list of possible positions together with the recommended move for each position.

In the case of 'Last One Wins - or does he?', positions such as:

```
7 A 0 0 (the start of a game with 7 counters)

7 A 2 1

7 A 5 3
```

are all different in the sense that the scores are different in each position. However the move recommended by a minimax lookahead will be the same in each case. Once a sequence of moves has been irrevocably made, there is nothing a player can do to change the points scored so far. All he can do is to optimise subsequent scoring from his point of view. Thus the best move for the player whose turn it is depends only an the number of counters left on the board. The following table lists the best move (in the minimax sense) for various positions:

counters left	best move: no. of counters to be taken
1	1
2	2
3	3
4	3
5	1
6	2
7	3
8	3
9	3
10	any move

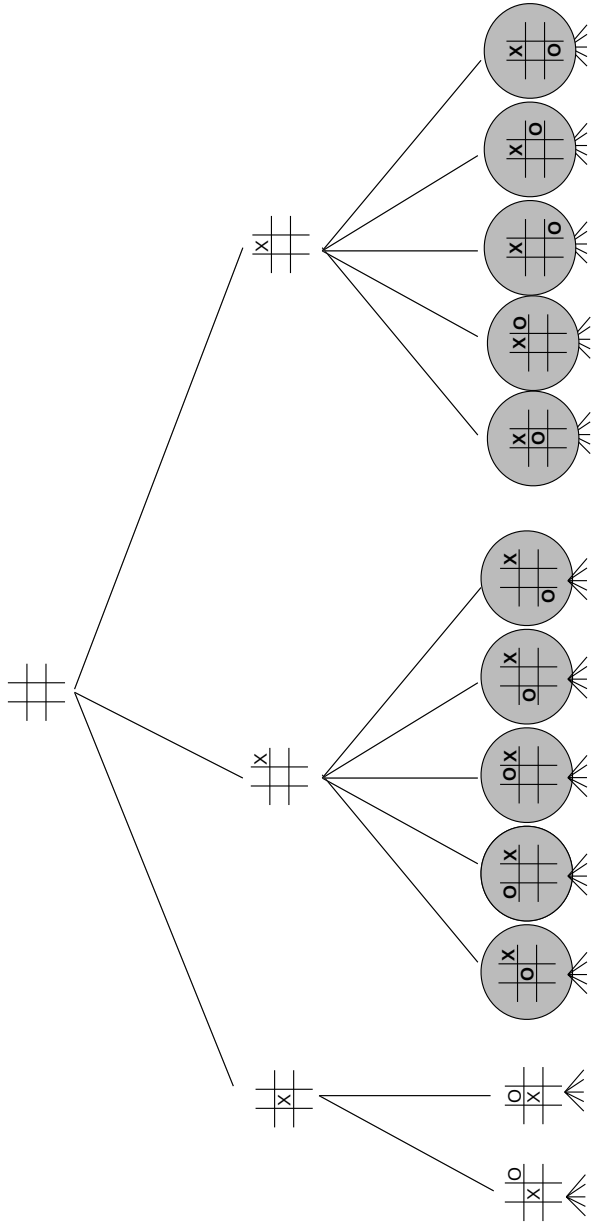
We could construct a table like this by hand for a game that we wish to program, but it would be more sensible to use a program to do it for us. Ways of using the minimax function previously written to automatically construct a decision table are discussed shortly. Whether we have constructed the above list by hand or automatically, our program can use it us a decision table as follows:

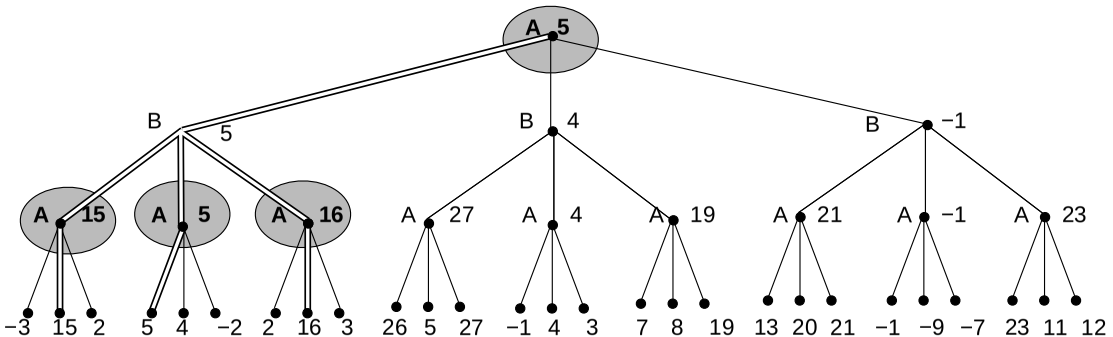
```
  5  DIM tablemove(10)
  6  PROCsetuptable
     :
 50  DEF PROCsetuptable
 51  LOCAL c
 52    FOR c = 1 TO 10
 53      READ tablemove(c)
 54    NEXT c
 55  ENDPROC
 56  DATA 1, 2, 3, 3, 1, 2, 3, 3, 3, 3
     .
     .
700  DEF PROCplayerA
710    move= tablemove(counters)
720    PRINT "I take "; move; " counters. "
730    counters = counters-move : Apoints = Apoints+move
740    turn$ = "B"
750  ENDPROC
```

In 'Last One Wins - or does he?', a game position to be looked up in the decision table consists of a single integer. The decision table can thus be represented by an array containing the recommended moves, a board position being used as a subscript to access the table. In a game with more complicated board positions, each board position together with its corresponding recommended move would have to be stored in the table. Table access techniques of the type discussed in Chapter 6 would then have to be used to find a given position in the table.

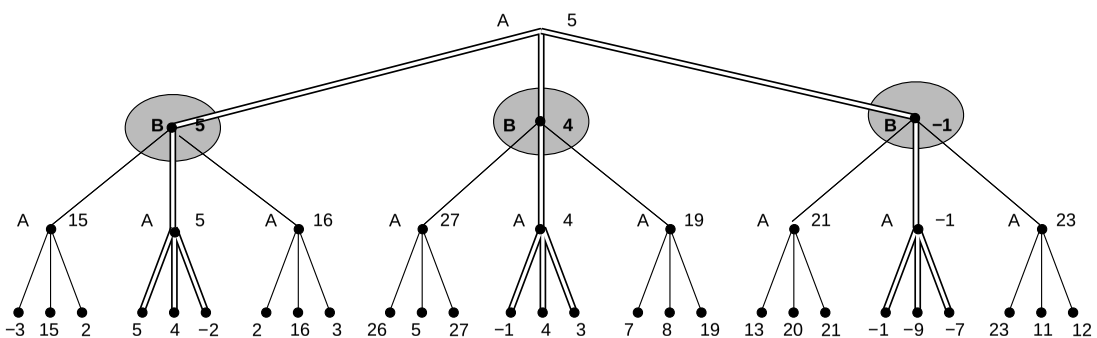
Whenever a decision table is being constructed for a game, spectacular savings in space requirements for the table can often be made by careful use of the fact that the decisions in the table will confine the game to a subtree of the main game tree. The diagram on the next page shows the first three levels of the game tree for Noughts and Crosses in which we have treated symmetrical positions as being identical. (As we have already mentioned, it is not easy for a program to recognise such symmetries.) If the program starts and the entry in the decision table for the initial position recommends playing in the centre, then the program will always do this and there is clearly no possibility that the program will ever have to make a move in any of the marked positions that are at depth two down the branches reached by the other possible initial moves. These positions can be left out of the decision table. Note that we cannot leave out <u>all</u> the positions down these branches of the tree. For example, if the program's opponent starts, then he might make any one of the three initial moves and thus all three possible positions at depth one have to be stored.

The next two trees illustrate, for a hypothetical game, the overall effect that we get if one of the two players always chooses his best move (in the minimax sense).





Best strategy subtree for player A



Best strategy subtree for player B

Best strategy subtrees for a hypothetical game

Note that although the minimu values have been calculated on the assumption that both players always make their best moves, we cannot rely on the program's opponent doing this. We must allow the possibility that the program's opponent will make a non-optimal move when it is his turn. If the program starts then play will be confined to one subtree (sometimes called a 'best strategy subtree') and if its opponent starts, then play will be confined to another subtree. These subtrees are marked with double lines in the diagrams. Only positions in these subtrees at which the program has to make a move (shaded) need be stored in a decision table. In this case we would need 7 decision table entries for a game in which there are 13 non-terminal positions. If the same analysis were applied to a game tree with 10000 non-terminal positions, the number of positions that would need to be stored in the decision table would be reduced to about 200. Clearly, we could not carry out such an analysis manually and we now introduce ways of doing it automatically.

Automatic construction of a decision table

The decision table that we constructed for 'Last One Win s-or does he?' could have been constructed automatically:

```
 50  DEFPROCsetuptable
 51  LOCAL counters
 52    Apoints=0 : Bpoints=0
 53    FOR counters = 1 TO 9 :REM 10 wont work
           (Too many FORs)
 54      tablemove(counters) = FNbestmove
 55    NEXT counters
 56  ENDPROC
```

This will take rather a long time to initialise the tables web time the program is used.

The program will exhibit rather more interesting behaviour if we initialise all the entries in the decision table to zero and extend FNbestmove as follows:

```
1100  DEF FNbestmove
1110  LOCALmove,movesavailable,max,nextval,bestvalsofar
1120    IF tablemove(counters)>0 THEN
          = tablemove(counters)
1130    PROCcheckmovesavailable
1140    max = -10
1150    FOR move = 1 TO movesavailable
1160      nextval = FNminval(counters-move,
            Apoints+move, Bpoints)
1170      IF nextval>max THEN
            max = nextval : bestmovesofar = move
1180    NEXT move
1190    tablemove(counters) = bestmovesofar
```

```
1195   = bestmovesofar
```

We must also revert to using the version of PROCplayerA that called FNbestmove. Each time FNbestmove is called, the decision table is examined, i an entry has previously been inserted for the given number of counters, then that entry determines the beat move. If no entry has yet been inserted, FNbestmove carries on and calculates the best move by carrying out a minimax analysis. Before this move is returned as a result of the function, it is inserted in the decision table for future use. This version of FNbestmove can now be used by PROCplayerA. Now consider what happens if we use the following loop to control the execution of our game-playing program.

```
10   REPEAT
11     PROCplaygame
13     INPUT "Another go (Y/N)", reply$
14   UNTIL reply$="N"
```

If we run the program and play several games during one run, then the program will gradually fill in entries in its decision table. For example, the first time the program has to choose a move with 10 counters on the board, it will take a long time to analyse the situation, but the next time, its response will be instantaneous - we have implemented a fairly primitive form of 'learning'. Furthermore, we are only inserting entries in the decision table for positions that are encountered during actual games. This, together with the effect discussed in the last section, would reduce the storage space required for a decision table in a game with more complicated board positions.

Another possibility that we shall not program in detail is some form of 'learning by trial and error'. We could set up a decision table in which each position has associated with it a set of values that determine the probabilities with which each move is to be selected. For a position that has not been encountered before, the probabilities for the moves available would all be equal. Thus the program would start by selecting completely random moves. During the course of a game the program would keep a record of the positions encountered and the moves selected. At the end of the game, if the program wins, then the probabilities of making all the moves recorded can be increased slightly; if the program loses, the probabilities for the moves made can be decreased slightly. The adjustments made must not more than slight: winning a game does not mean that all the moves made were good. Over a period, the performance of the program will gradually improve.

If we are experimenting with the 'learning' techniques discussed in this section, then it can be very frustrating

if we have painstakingly improved the program's performance during a RUN and then have to start from scratch again the next day. For this reason, a useful option in such a program would be the ability to output a decision table to a file and to read the table from the file at the start of the next run.

Playing by rule

For simple games, it is often possible to define a rule or rules that can be used to select a good move without exhaustive exploration of the game tree.

In the game 'Last One Wins' used at the end of Chapter 1, there is a very simple rule that can be used to find a winning move if there is one available:

> Try to make a move that leaves a multiple of 4 counters on the board. (If no such move is available, then you are in a losing position.)

In Noughts and Crosses, we might use a sequence of rules that are to be applied in turn until a move is found. For example,

> If there is a winning move, make it.
> If the opponent has a winning move, block it.
> If the centre is empty, play there.
> Make a random move.

This Noughts and Crosses strategy does not always select the best move and draws attention to the fact that a set of rules need not be optimal. In fact, it would be quite easy to define a set of simple rules for choosing a move in Chess, but the resulting program would play a very poor game.

Exercises

1 Extend your Noughts and Crosses program so that it chooses a move by using a set of simple rules like those outlined above.