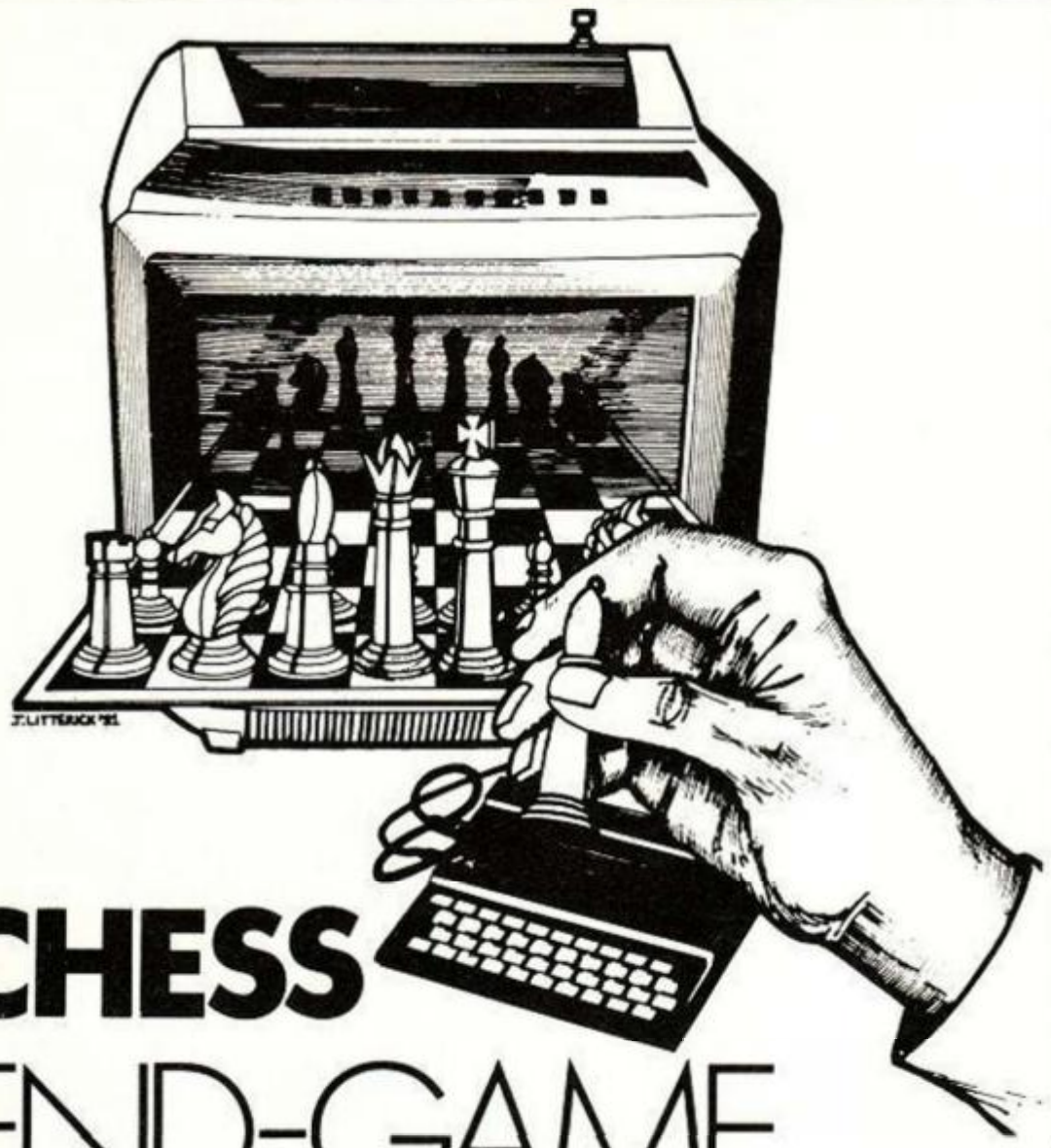


CHESS END-GAME

Exotica such as the king ripple and the pawn-advance routine are two of the techniques John White has incorporated into his entertaining chess program in Basic, End-Game.

END-GAME has been written in Basic to complement the draughts program J-Checkers, published in the October issue of *Your Computer*. It exemplifies the method of move assessment known as iterative deepening.

I have chosen the end-game of chess as a model because it limits the number of pieces used and because the concept of mobility — essential in full games of chess — can, at a pinch, be ignored to keep the time taken for the game within manageable limits. I have eschewed fancy time- or memory-saving tricks for clarity.

Having tested this program, I have satisfied myself that it is not possible to write a satisfactory program for playing a chess end-game using a look-ahead of only two-ply. I hope this information will be of use to those contemplating writing their own chess programs.

End-Game does, however, play a fractionally more sensible chess end-game than many of the weaker chess computers available commercially, bearing in mind the fact that a compiled version would run in about two seconds. The interpreted Basic version presented here requires an average of two minutes a move.

The end-game of chess is hard for a human to play well, but very difficult indeed for a chess computer. A human can easily see at a glance what will happen six to seven moves ahead for both sides — grandmasters can see much, much more.

A chess computer will normally only analyse two or three moves ahead — four- to six-ply — although one or two of the most modern machines switch in extra routines for the end-game when sufficiently little material remains on the board. Under these circumstances up to five moves ahead — 10-ply — may be evaluated. Even so, the play is still weak by human standards. The classic problem is that shown in figure 1.

It is possible for a human to see at once that black's only sensible move is K-B6 — or B8 or B7. Anything else loses the pawn to white's attacking king. I shall avoid the problem of whether black can win even if he does save the pawn. Yet very few chess computers can see this solution, and most play pawn endings very badly, moving pieces almost at random.

Since the necessary deep search to play a good end-game is very time-consuming, I have tried in End-Game to produce an evaluation function which will play a recognisable end-game superior to that of most chess computers but using only a two-ply search. Essentially I have relied on the well-known maxim of "Push a passed pawn".

End-Game is written in Basic which imposes its own stunning restriction on what can be placed in the program: interpreted Basic runs some 200 times more slowly than the machine code used in chess computers and a complete game of chess is out of the question. Restricting the pieces to pawns and king only gives a respectable game with a clear objective: advancement of a pawn to the eighth rank.

The first player to do this has essentially won at chess, and has won End-Game outright. It may be noted that the powerful Sargon 2.5 and Morphy chess programs also adopt this policy in their end-game play, and will make any sacrifice to delay the arrival of an enemy pawn on the eighth rank.

End-Game uses a single subroutine to evaluate the position arising after each move — instead of evaluating the merit of each move itself, a strategy employed in other published games. The moves of each piece are generated by the program which assigns a score to the position arising from each move at the first level of search — one-ply.

The moves are then sorted, using a fast-sort routine which arranges the score in order of decreasing merit. The moves creating the scores are also rearranged, of course.

The program now calls itself — an example of recursion in Basic — to generate the

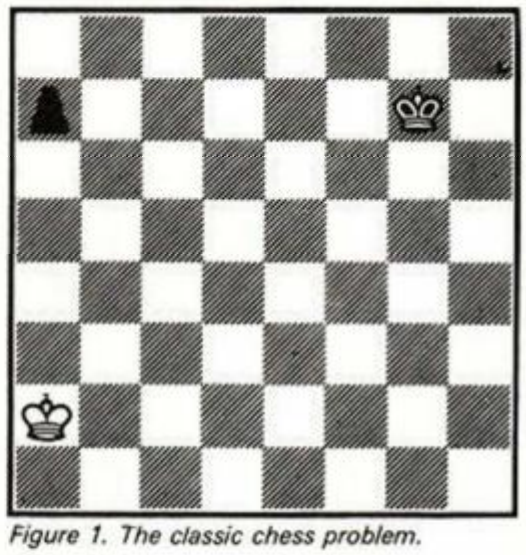


Figure 1. The classic chess problem.

responses to its sorted moves. It assumes that the opponent will be trying to maximise his score, and thus minimise the machine's score. So the best — lowest-scoring — opponent move is stored in location.

This is combined with the first-ply score and compared with the highest total yet found for a program move, which is stored in location R(0). R(0) is continually updated as better moves are found for the machine — moves for which the opponent can find only weak responses.

An important feature of this search is the co-called "alpha-beta pruning". If any opponent response makes the machine's move under consideration worse than a previous stored machine's move, then there is no need for the machine to consider any further responses by the opponent to the machine move under consideration. The flag "AB" is set to 1, which stops any further searching of that move.

Alpha-beta pruning can save a good deal of unnecessary searching and thus a great deal of time. It is widely used in chess computers today. To be most effective, it is best to consider the most-likely-best machine move first, and also the most-likely-best opponent response.

The most-likely-best machine move has been derived by the sort which we considered earlier. The most-likely-best opponent move is hard to determine without going through them all — which, of course, defeats the whole point of alpha-beta searching.

I have instead adopted what I believe to be a novel heuristic: the best response found for the opponent for the previous machine move is evaluated first for the next machine move. This has proved to be quite effective at saving time in End-Game.

The nett effect of alpha-beta pruning on End-Game is quite spectacular in reducing response time. Anyone who doubts this should try deleting line 800.

The whole process I have described — move generation, sorting, counter-move generation, alpha-beta pruning — is known as iterative deepening. It will be appreciated that the best move so far found is always available, and machines using this technique generally display this best-move-yet — a feature which I have emulated in End-Game.

Many chess computers employ an adjustable timer which will interrupt the machine and display the best-move-yet as its move — examples include the Super System III — while others carry the process to its logical conclusion — examples are Sargon 2.5 and End-Game.

Because of the time restrictions imposed by interpreted Basic, End-Game evaluates the material and strategic position for both sides just once. Captures and certain other strong moves are evaluated for material gains at a further two levels.

The form of End-Game has been dictated by attempts to increase the speed. Constant calling of subroutines looks very pretty, but tends to slow execution time, while writing the same thing out several times is faster, but uses far more memory. I have stacked the most commonly-used subroutines at the head of the program to speed up their location when called.

The greatest retardation of any Basic program is caused by the dreaded If statement. When this occurs in a loop, the loss of time accelerates rapidly. The evaluation function is called after every potential move, yet If statements are essential in it if it is to serve any useful purpose. I have moved some of the evaluation features from the main evaluation subroutine to reduce the number of times they are called.

It is interesting to see how careful selection of moves can reduce total thinking time for the machine. Lines 130-170 are called every time a pawn move is considerd and should, one might think, slow the program compared with the speed of execution without these lines which test to see if advancing a pawn enables the opponent to snap it up immediately.

After all, the second level of search will find that the pawn can be captured by a strong opponent move, so why put it in? In fact, End-Game likes to advance pawns and so, by deterring an advance into the jaws of an opponent, a more sensible first move is put at the top of the list after the sort.

Thereafter, alpha-beta pruning does its work and the weak pawn advance is barely considered instead of being fully evaluated as the first move on the list. This saves a great deal of time. Thus the nett effect of the time-consuming lines 130-170 is actually to accelerate the program.

Variables defined in program lines 1110-1120:
CC = 0.012 CE = 0.1 CF = 0.2 CG = 0.3 CH = 10
CJ = 30 CK = 15 CL = 50 CM = 10 CQ = 2 CD = 1
CZ = 3

Material count: pawn = CD king = CZ	
Pawn moves:	
Do not approach enemy king	−CG
Do not approach enemy pawn	−CG
Stay off edge of board	−1/CH
Advance to rank Y	$+Y^2 \times CC$
Avoid having Y pawns on one file	$-(Y-1) \times Y \times CF$
Pawn advance: no opposition in first channel	+CG
to eighth rank	+0.5
score for first channel	Material count/CH
score for second channel	Material count/CM
En passant threat	−0.8
King moves:	
King opposition	+CE
King environment	+ 1/CK, + 3/CK
Avoid capture by pawn	−5
Do not stray from centres squares	+1/CJ
King ripple	Material count/(CQ × CL)

Table 1. Evaluation table for End-Game.

The evaluation features are listed in table 1. The variables which store the scores for different features, shown in table 1, are all found in lines 1110 and 1120, and so can be altered if you feel like experimenting.

Two features which I believe to be original are the pawn-advance routine — subroutine 510 — and the king ripple — lines 2260-2290. Both are stored outside the main evaluation subroutine.

The pawn-advance examines a three-square-wide channel ahead of the pawn after it has moved, all the way to the eighth rank. The move is scored according to whether the channel is obstructed — enemy piece in front, king scores high — or assisted — friendly piece in front, king scores high. The same channel is then examined again for its entire length, and again scored.

The second score shows whether the advancing pawn has numerical supremacy over the opposition: that is, one of two pawns will be encouraged to advance if the path is blocked by only one enemy pawn.

Obviously, this is a very crude evaluation feature, but it works relatively well for End-Game while minimising the number of If statements required.

The king ripple is a very low-scoring feature put in solely to prevent the king wandering aimlessly when most of the other material has been removed from the board. All the squares at a distance of two squares from the king, then three, then four and so on, are examined until another piece of either side has been found. The king then heads towards this piece.

King ripple is time-consuming and is evaluated only for the computer's pieces. Coupled with the routine which weakly discourages the king from wandering outside the central 16 squares, it should prevent the king from becoming "lost" for too long.

King environment searches each square within one move of the king, and scores favourably — +3/15 — for each enemy pawn so located and less favourably — +1/15 — for each friendly pawn. Obviously, the two kings cannot approach each other.

Other evaluation features include low scoring for pawns on either edge of the board, avoidance of doubled pawns — trebled or quadrupled pawns are punished exponentially — an exponentially-increasing score as a pawn advances to the eighth rank and encouragement for one king holding the opposition over the other.

I have remembered End-Game's chess origins by not insisting that the machine advance a pawn to the eighth rank if a good move, such as a capture, exists elsewhere on the board.

En passant has been catered for by a somewhat elementary method. If the human makes a move which enables the machine to capture *en passant*, the capture is given priority and properly evaluated. However, the machine does not allow for *en passant* when otherwise evaluating moves: instead, the possibility of *en passant* is assigned a score of "undesirable" without evaluating in depth.

End-Game was written in standard Microsoft Basic with no Peeks or Pokes. The use of cursor commands, including screen clear and home greatly improves display.

Lines 1310 and 1770 operate a timer routine for my Sharp MZ-80K and can be adapted or ignored. Many computers do not like jumping from loops, which has influenced some of my program lines. Other Sharp users will require one of the Basic Extensions for the logical

(continued on next page)

A B

Pawn advance. A shows first channel, B second channel.

Pawn advance. The most likely move for black will be F,7 — F,6.

King ripple.

A B C

King environment. The most favoured position for the king is C; less favourable is B and least favourable is A.

A starting position for games pre-stored in End-Game.

(continued from previous page)

operators And and Or in some lines and the string inequalities in others.

Line 500 returns a value of −1 for each bracketed statement which is true, and 0 if false. This line runs some 20 percent faster than the corresponding If statements would.

Sadly, the program runs to 9.5K as it stands. This can be trimmed to 8K by removing the fast sort — this will slow it somewhat — by removing the screen-display lines, and by removing all but two of the data statements, together with the lines which select the data statements.

Program Function	Line
Evaluation	240-400
King environment	410-500
Pawn advance	510-629
Move storage	630-670
Third-level captures	680-810
Fourth-level captures	820-920
Data statements	930-1040
Variables defined for evaluation	1110-1120
Set-up position	1130-1280
First ply	1290-1540
Second ply	1550-1760
Move display	1770-1820
Input moves	1830-2080
King-move generator	2090-2310
Pawn one-move generator	2320-2350
Pawn two-move generator	2360-2450
Pawn capture	2460-2550
Fast sort	2560-2830
Alpha-numeric conversion	2840-2870
Screen display	2880-3010

End-Game has six different games prestored in 12 Data statements — the starting positions are shown in the diagrams. These can be selected, or the program will choose randomly between them. A display of the board is given — copy it on to your chess board. To set up your own position, it will be necessary to alter two Data statements.

The machine will prompt

YOUR MOVE

when read, followed by

FROM?

It will now accept ordinary alpha-numeric entries, such as

(FROM) D,2 (TO) D,4

Alternatively, typing P,1 will give a display of the board which is displayed only if you ask for it. Typing Q,1 will reveal what the machine thinks your move should be and typing Y,1 will cause the machine to act on its own suggestion for your move without need to enter it.

The only error check run by the machine on your input is that there is a piece of yours at the point from which you are trying to move. Thus you can move pieces round both easily and illegally should you want to.

```
99 REM**END-GAME by J.F.White.
100 PRINT[CLS];GOTO1050
110 GOSUB230
120 GOSUB510:REM**CAN PAWN ADVANCE?
130 FORPJ=-1TO1
140 IFA(X+PJ,Y-AO)=-CZ*AOTHENO=O-AO*CG
150 IFPJ=0THEN170
160 IFA(X+PJ,Y-AO)=-AOTHENO=O-AO*CG
170 NEXT
180 N=N+1
190 GOSUB630
200 GOSUB680
210 GOSUB1480
220 RETURN
230 REM**EVALUATION
240 RX=A(X,Y):A(X,Y)=A(I,J):A(I,J)=0
250 O=0
260 FORIZ=1TO8
270 IFA(I,IZ)=AO OR A(8,IZ)=AOTHENO=O-AO/CH
280 FORJZ=1TO8
290 O=O+A(IZ,JZ)
300 IFA(IZ,JZ)=0THEN340
310 IFA(IZ,JZ)=CDTHENO=O+(9-JZ)*(9-JZ)*CC:MA=MA+1:GOTO340
320 IFA(IZ,JZ) =AO*CZGOSUB410:GOTO340
330 IFA(IZ,JZ)=-CDTHENO=O-JZ*JZ*CC:MB=MB+1
340 NEXT
350 O=O-((MA-CD)*CF*MA)
360 O=O+((MB-CD)*CF*MB)
370 MA=0:MB=0
380 NEXT
390 A(I,J)=A(X,Y):A(X,Y)=RX
400 RETURN
410 FORP3=-2TO2STEP4
420 IFIZ+PJ<1ORIZ+PJ>8ORJZ+P3<1ORJZ+PJ>8THEN440
430 IFA(IZ+PJ,JZ)=-CZ*AOORA(IZ,JZ+PJ)=-CZ*AOTHENO=O+AO*CE:GOTO440
440 NEXT
450 FORI7=-1TO1:FORJ7=-1TO1
460 IFA(IZ+I7,JZ+J7)=0THEN480
470 O=O-((A(IZ+I7,JZ+J7)-2*AO)*CD/CK)
480 NEXT:NEXT:O=O+3*AO/CK
490 IFA(IZ-CD,JZ-AO)=-AOORA(IZ-CD,JZ-AO)=-AOTHENO=O-5*AO
500 O=O+((IZ<3)+(IZ>6)+(JZ<3)+(JZ>6))*AO/CJ:RETURN
510 FORP2=-1TO1:IFP2+X>8THEN580
520 FORO2=Y-AOTO(4-4*AO)STEP-AO
530 OP=OP+A(P2+X,O2)
540 NEXT
550 FORO2=1TO8
560 OO=OO+A(P2+X,O2)
570 NEXT
580 NEXT
590 IFOP=0THENO=O+CG*AO
600 IFY=1THENO=O+.5
610 O=O+OP*CD/CH+OO/CM:OP=0:OO=0
620 RETURN
630 REM**MOVE STORAGE
640 AA(N)=X:BB(N)=Y
650 AB(N)=I:BC(N)=J
660 D(N)=O
670 RETURN
680 REM**EVALUATE CAPTURES
690 IFDA(K)-D(N)>.8OANDFLAG=1THEN710
700 GOTO790
710 REM**3RD LEVEL
720 FORI5=-1TO1:FORJ5=-1TO1
730 IFI5=0ANDJ5=0THEN750
740 IFA(AA(N)+I5,BB(N)+J5)=CZ*-AOTHEND(N)=D(N)-A(AB(N),BC(N)):GOSUB830:NG=1
750 NEXT:NEXT
760 IFNG=1THENNG=0:GOTO790
770 IFA(AA(N)+AO,BB(N)-AO)=-AOTHEND(N)=D(N)-A(AB(N),BC(N)):GOSUB830:GOTO790
780 IFA(AA(N)-AO,BB(N)-AO)=-AOTHEND(N)=D(N)-A(AB(N),BC(N)):GOSUB830
790 IFD(N)<SOTHENSO=D(N):S1=AA(N):S2=BB(N):S3=AB(N):S4=BC(N)
800 IFDA(K)+SO<=R(O)THENAB=1
810 RETURN
820 REM**FOURTH LEVEL
830 FORI4=-1TO1:FORJ4=-1TO1
840 X2=AA(N)+I4:Y2=BB(N)+J4
850 IFX2=AB(N)ORY2=BC(N)THEN870
860 IFA(X2,Y2)=CZ*AOTHEND(N)=D(N)+AO:NG=1
870 NEXT:NEXT:IFNO=1THENNO=0:RETURN
880 IFAA(N)-1=AB(N)THEN910
890 IFA(AA(N)-1,BB(N)-1)=AOTHEND(N)=D(N)+AO:RETURN
900 IFAA(N)+1=AB(N)THEN920
910 IFA(AA(N)+1,BB(N)-1)=AOTHENO(N)=D(N)+AO
920 RETURN
930 DATA4,5,3,2,6,1,3,6,1,4,7,1,5,6,1,6,6,1
940 DATA 6,4,-3,2,3,-1,3,3,-1,4,2,-1,5,3,-1,6,3,-1
950 DATA 4,6,3,1,7,1,2,7,1,3,7,1,7,7,1,8,7,1
960 DATA7,3,-3,1,2,-1,2,2,-1,3,2,-1,6,2,-1,8,3,-1
970 DATA4,6,3,1,5,1,2,7,1,8,7,1,0,0,0,0,0,0
980 DATA4,4,-3,2,3,-1,7,3,-1,8,3,-1,0,0,0,0,0,0
990 DATA4,8,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1000 DATA4,4,-3,4,2,-1,0,0,0,0,0,0,0,0,0,0,0,0
1010 DATA7,6,3,8,6,1,0,0,0,0,0,0,0,0,0,0,0,0
1020 DATA6,3,-3,7,3,-1,8,3,-1,0,0,0,0,0,0,0,0,0
1030 DATA4,6,3,2,6,1,3,5,1,5,6,1,7,7,1,8,6,1
1040 DATA5,2,-3,1,2,-1,3,4,-1,4,2,-1,7,3,-1,8,3,-1
1050 DIMAA(25),BB(25),D(25),A(20),B(20)
1060 DIMC(20),D(20)
1070 DIMA(9,9)
1080 DIMAB(25),BC(25),DA(25)
1090 DIMR(4),S(4),R$(4),S$(4)
1100 DIMB(INT(LOG(25)/LOG(2)+1),2)
1110 CC=.012:CD=1:CE=.10:CF=.2:CG=.3:CH=10: REM **VARIABLES IN EVALUATION
1120 CI=10:CJ=30:CK=15:CL=050:CM=10:CZ=3
1130 PRINT:PRINT:PRINT
1140 PRINT"SELECT POSITION(1-6) OR RND"
1150 INPUTA$:IFA$="RND"THENRN=INT(RND(1)*6)+1:GOTO1170
1160 RN=VAL(A$):IFRN<1ORRN>6THEN1140
1170 PRINT[CLS][8CD][5CR]"END - GAME by J.F.White."
1180 FORIA=1TORN
1190 FORI=1TO8:FORJ=1TO8:A(I,J)=0:NEXT:NEXT
1200 FORI=1TO16:IFI<11ANDI>6THEN1220
1210 READA(I),B(I),A(A(I),B(I))
1220 NEXT
1230 NEXT
1240 RESTORE
1250 GOSUB2880
1260 PRINT:PRINT"DO YOU WANT TO GO FIRST(Y\N)?"
1270 INPUTA$:IFA$="Y"THENPRINT:PRINT:GOTO1830
1280 IFA$<>"N"THEN1260
1290 PRINT[CLS][2CD]"   FROM    .    TO    ."
1300 TO=-100
1310 TI$="000000"
1320 R(0)=-100
1330 AO=1:SO=99:AB=0
1340 FLAG=0
1350 GOTO1390
1360 I=S(3):J=S(4):SO=99:AB=0
1365 IFK=1THEN1390
1370 IFA(I,J)=AOGOSUB2330
1380 IFA(I,J)=CZ*AOGOSUB2090
1390 FORI=1TO8:FORJ=1TO8
1400 IFAB=1THEN1430
1410 IFA(I,J)=CZ*AOGOSUB2090
1420 IFA(I,J)=AOGOSUB2330
1430 NEXT:NEXT
1440 IFN<>0THEN1540
1450 PRINT:PRINT:PRINTTAB(7);"STALEMATE !":PRINT:END
1480 IFFLAG=1ORN=0THEN1530
1490 IFD(N)>TOTHENTO=D(N):T1=AA(N):T2=BB(N):T3=AB(N):T4=BC(N):GOTO1510
1500 GOTO1530
1510 T1$=CHR$(ASC(STR$(T1))+16):T3$=CHR$(ASC(STR$(T3))+16)
1520 PRINT[HOME][2CD]";TAB(9);T3$;TAB(12);T4;TAB(20);T1$;TAB(23);T2
1530 RETURN
1540 IFFLAG=1THENRETURN
1550 REM**SORT
1560 M=N:FLAG=1:AO=-1
1570 GOSUB2560
1580 REM**2ND LEVEL
1590 R(0)=-99
1600 FORK=1TOM
1610 A(M-K+1)=AA(K):B(M-K+1)=BB(K):C(M-K+1)=AB(K):D(M-K+1)=BC(K)
1620 DA(M-K+1)=D(K):D(K)=0
1630 AA(K)=0:AB(K)=0:BC(K)=0:BB(K)=0
1640 NEXT
1650 FORK=1TOM
1660 N=0
1670 RY=A(A(K),B(K)):A(A(K),B(K))=A(C(K),D(K)):A(C(K),D(K))=0
1680 GOSUB1360
1690 OZ=SO+DA(K)
1700 IFOZ>R(O)THENR(O)=OZ:R(1)=A(K):R(2)=B(K):R(3)=C(K):R(4)=D(K):GOTO1720
1710 GOTO1750
1720 S(1)=S1:S(2)=S2:S(3)=S3:S(4)=S4
1730 GOSUB2840
1740 PRINT[HOME][2CD]TAB(9);R$(3);TAB(12);R(4);TAB(20);R$(1);TAB(23);R(2)
1750 A(C(K),D(K))=A(A(K),B(K)):A(A(K),B(K))=RY
1760 NEXT
1770 PRINT"MY TIME FOR THIS MOVE = ";MID$(TI$,3,2);" MINS ";RIGHT$(TI$,2);
1772 PRINT" SECS."
1780 GOSUB2840
1790 PRINT[3CD][2CR]"FROM ";R$(3);" ,";R(4);" TO ";R$(1);" ,";R(2)
1800 IFR(2)=1 ANDA(R(3),R(4))=1THENPRINT:PRINT:PRINT[8CR]"I WON!":END
1810 IFABS(R(1)-R(3))=1ANDR(2)=3ANDA2=R(1)ANDB2=R(2)+1THENA(A2,B2)=0
1820 A(R(1),R(2))=A(R(3),R(4)):A(R(3),R(4))=0
1830 PRINT:PRINT"YOUR MOVE? " ;FLAG=0:N=0:TO=-100
1840 PRINT: INPUT"FROM  "; A1$,B1$:B1=VAL(B1$)
1842 IFA1$="Y"THENA1$=S$(3):B1=S(4):A2$=S$(1):B2=S(2):GOTO1900
1850 IFA1$="P"THENGOSUB2800:GOTO1830
1860 IFA1$<>"Q"THEN1890
1870 PRINT"YOUR BEST REPLY IS ";S$(3);" ,";S(4);" TO ";S$(1);" ,";S(2)
1880 GOTO1830
1890 INPUT"TO  ";A2$,B2$:B2=VAL(B2$)
1900 A1=VAL(CHR$(ASC(A1$)-16))
1910 A2=VAL(CHR$(ASC(A2$)-16))
1920 IFA(A1,B1)>-1THENPRINT"ERROR - TRY AGAIN":GOTO1830
1930 IFB2=BANDA(A1,B1)=-1THENPRINT:PRINT:PRINT[3CR]"YOU WON!":PRINT:END
```

```
1940 PRINT[CLS][2CD]"   FROM    ,    TO    , "
1950 A(A2,B2)=A(A1,B1):A(A1,B1)=0
1960 IFA(A2,B2)=-1ANDB1=5THEN1980
1970 GOTO1990
1980 IFABS(A1-A2)=1ANDR(2)-R(4)=-2ANDR(1)-A2=0THENA(A2,B2-1)=0
1990 IFB1=2ANDB2=4THENA0=1:GOTO2010
2000 GOTO2080
2010 IFA(A2+1,B2)=1THENI=A2+1:J=B2:N=1:GOSUB2040
2020 IFA(A2-1,B2)=1THENI=A2-1:J=B2:N=1:GOSUB2040
2030 GOTO2080
2040 A(A2,B2-1)=A(A2,B2):A(A2,B2)=0:GOSUB2470:A(A2,B2)=A(A2,B2-1)
2050 A(A2,B2-1)=0
2060 T0=0(N)
2070 RETURN
2080 GOTO1510
2090 REM***KING MOVE
2100 FORII=-1TO1:FORIJ=-1TO1
2110 IFAB=1THEN2300
2120 IFII=0ANDIJ=0THEN2300
2130 X=I+II:Y=J+IJ
2140 IFX <1ORX >8ORY <1ORY >8THEN2300
2150 IFA(X,Y)=AOTHEN2300
2160 FORP4=-1TO1:FORO4=-1TO1
2170 IFA(X+P4,Y+O4)=-C2*AOTHENNO=1
2180 IFA(X-1,Y-A0)=-AOTHENNO=1
2190 IFA(X+1,Y-A0)=-AOTHENNO=1
2200 NEXT:NEXT:IFNO=1THENNO=0:GOTO2300
2210 GOSUB230
2220 N=N+1
2230 GOSUB630
2240 GOSUB680
2245 IFFLAG=1THEN2300
2250 OO=0:CO=2
2252 FORP3=-COTOCO
2254 IFX+P3<1ORX+P3>8THEN2264
2256 IFY+CO>8THEN2260
2258 OO=OO+ABS(A(X+P3,Y+CO))
2260 IFY-CO<1THEN2264
2262 OO=OO+ABS(A(X+P3,Y-CO))
2264 IFY+P3<1ORY+P3>8THEN2274
2266 IFX-CO<1THEN2270
2268 OO=OO+ABS(A(X-CO,Y+P3))
2270 IFX+CO>8THEN2274
2272 OO=OO+ABS(A(X+CO,Y+P3))
2274 NEXT
2276 IFOO>0THEN0(N)=0(N)+OO/(CO*CL):GOTO2290
2278 CO=CO+1:IFCO>8THEN2290
2280 GOTO2252
2290 GOSUB1480
2300 NEXT:NEXT
2310 RETURN
2320 REM**PAWN 1-MOVE
2330 X=I:Y=J-AO
2340 IFA(X,Y)<>0THEN2460
2350 GOSUB110
2360 REM**PAWN 2-MOVE
2370 IF(AO=1)AND(J=7)THEN2400
2380 IF(AO=-1)AND(J=2)THEN2400
2390 GOTO2460
2400 X=I:Y=J-2*AO
2410 IFA(X,Y)<>0THEN2460
2420 GOSUB110
2425 REM**EN PASSANT THREAT
2430 IFA(X+1,5+(AO=-CD))=-AOTHEN0(N)=0(N)-AO*.8
2440 IFA(X-1,5+(AO=-CD))=-AOTHEN0(N)=0(N)-AO*.8
2450 GOSUB1480
2460 REM** PAWN CAPTURE
2470 X=I-1:Y=J-AO
2480 IF(AO=1)AND(A(X,Y)<0)THEN2510
2490 IF(AO=-1)AND(A(X,Y)>0)THEN2510
2500 GOSUB110
2510 X=I+1:Y=J-AO
2520 IF(AO=1)AND(A(X,Y)<0)THEN2550
2530 IF(AO=-1)AND(A(X,Y)>0)THEN2550
2540 GOSUB110
2550 RETURN
2560 REM**QUICK SORT
2570 SS=1
2580 B(1,1)=1:B(1,2)=M
2590 LL=B(SS,1):RR=B(SS,2):SS=SS-1
2600 II=LL:JJ=RR:XX=0(INT(RND(1)*(RR-LL)+.5)+LL)
2610 IF0(II)>=XXTHEN2630
2620 II=II+1:GOTO2610
2630 IFXX>=0(JJ)THEN2650
2640 JJ=JJ-1:GOTO2630
2650 IFII>JJTHEN2720
2660 WW=0(II):0(II)=0(JJ):0(JJ)=WW
2670 WW=AA(II):AA(II)=AA(JJ):AA(JJ)=WW
2680 WW=BB(II):BB(II)=BB(JJ):BB(JJ)=WW
2690 WW=AB(II):AB(II)=AB(JJ):AB(JJ)=WW
2700 WW=BC(II):BC(II)=BC(JJ):BC(JJ)=WW
2710 II=II+1:JJ=JJ-1
2720 IFII<=JJTHEN2610
2730 IFJJ-LL>=RR-IITHEN2770
2740 IFII>=RRTHEN2760
2750 SS=SS+1:B(SS,1)=II:B(SS,2)=RR
2760 RR=JJ:GOTO2800
2770 IFLL>=JJTHEN2790
2780 SS=SS+1:B(SS,1)=LL:B(SS,2)=JJ
2790 LL=II
2800 IFLL<RRTHEN2600
2810 IFSS>0THEN2590
2820 PRINT"SORT COMPLETE"
2830 RETURN
2840 FORK=1TO2STEP2
2850 R$(K)=CHR$(ASC(STR$(R(K)))+16):S$(K)=CHR$(ASC(STR$(S(K)))+16)
2860 NEXT
2870 RETURN
2880 REM**SCREEN DISPLAY OF POSITION
2890 PRINT[CLS]
2900 FORJ=8TO1STEP-1:FORI=1TO8
2910 PRINTTAB(5*I-5);
2920 IFA(I,J)=0THENPRINT".";
2930 IFA(I,J)=CDTHENPRINT"P";
2940 IFA(I,J)=-CDTHENPRINT"P*";
2950 IFA(I,J)=CZTHENPRINT"K";
2960 IFA(I,J)=-CZTHENPRINT"K*";
2970 NEXT:PRINT J ;PRINT:NEXT
2980 PRINT:PRINT"A   B   C   D   E   F   G   H"
2990 PRINT:PRINT:PRINT"PRESS ANY KEY TO CONTINUE"
3000 GET A$:IFA$=""THEN3000
3010 RETURN
```

Drawing on examples from his program written for the Sinclair machines, Philip Joy shows you how to go about creating a chess game of your own.

Writing chess

THE ZX-80 IS not the best machine on which to write long and complex programs because of its Basic. There are hardware problems as well as software ones — a poor keyboard for entering code in quantity, and slow speed. The Basic does not let you use two-dimensional arrays — which at first might seem a problem — because of an eight-by-eight board.

However, it proves not to be, and in fact it helped me to such an extent that I have kept a 64 array for my version for the ZX-81, even though it has multi-dimensional arrays. By having integer arithmetic only, the troubles with many INTs were overcome. The main hardship is the fact that it did not have a really easy way to enter machine code. Read or Data would have helped or even a monitor would have made the entry of machine code easier.

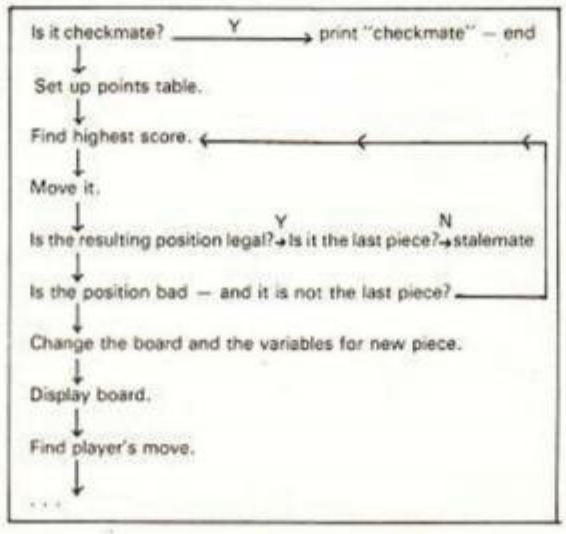
As the program neared completion, it was structured around five main units: initialisation, movement, points, player, and back-up. Each one of these units has a specific use and place in the program. The movement was the most difficult to write and proved to have a great deal of bugs. The points section is the thinking part of the program and calculates the best moves: it was the easiest part to write.

The player is the unit which keeps the user informed and sorts his moves for the computer to respond to. Back-up plays the greatest part in keeping the program working smoothly. It finds, for example, the level of play, sorts the points and deals with the machine code.

The computer obeys the laws of the game in a much more logical way than the average player might. If a good situation arises, a player might rush his next move and make either an illegal move, or one which could lead to his losing the game. If a computer obeys the laws of chess, checking for such things as discovered check, illegal castling, and general illegal piece moves, it does mean that this element of rush is removed.

Most of the moves are straightforward or are mixtures of two simple moves, e.g., the queen's move. The bishop and the rook move-

Figure 1. How the computer makes a move.



ments could be calculated by a person with some experience of computer programming. With some thought, the L-shaped knight movement and the queen's could be solved. However, the two pieces, the king and the pawn which move only one square at a time, are the pieces with many conditions attached to them. These two pieces have many characteristics which, although are not directly connected with them, make the movement hard to perfect.

The program is based around a 64-character array which holds all the black and white pieces and the blank spaces. The program uses two vectors to search for its move. One is the horizontal vector, or the number of squares down, while the vertical or the number of squares along is used as the other vector. To look into the array, these two vectors are brought together by the equation

$$(x-1)*8+y$$

to give the square about which we want to find some information. Our points table is scanned for a move, and this move is then tried. If the move fails, it is given a value of 0 in the points table to prevent its being tried again. When a move is found, the array is updated, and the board is printed for the player's turn. In any move, the important thing to remember is that the moving of a piece can cause check, or checkmate. Many human players could overlook this part of the game.

(continued on next page)

(continued from previous page)

After each move, the computer will discover whether its king is in check. If it is, the position is restored and another move tried. If the same directions for a piece were tried in the same sequence every time, the bishop, for example, would always move upwards and left. The computer must, therefore, try different directions in a different order each time.

Difficult decisions

The initialisation stage is the section where the most difficult decisions are made. For example, you have to choose the form of storage, the arrays and their sizes and the different variables. The representation of the pieces, and the style of play will all be affected by the way you decide. It is also here where you can include features which are not needed or could be combined to reduce space, and time.

The movement of the pieces is the only part of the program which has been heavily flow-charted. It is logical and is a part humans do not consider in great detail — hence the danger of bugs. It took about a month to remove the bugs in this section, mostly by trial and error.

We search for a possible move by working across the board and using our memory to decide whether a piece can move or not. The computer discovers a move by trial and error. We must also be careful to take into account that the program could be faced with a situation for which it cannot find a move. In this position, the computer will have to decide whether it is checkmate or stalemate.

The pawn was the easiest to tackle with its one move forward or two if it was on its first move. All you have to do is to subtract from the horizontal vector and check whether the piece can move to this square. If it is on row 7 or row 2 depending on colour, we can take two from the horizontal vector. However, before we move we can check whether a diagonal — found by adding or subtracting one from both the vectors, depending on which way you are going — is occupied by an enemy piece. If it is, we would take it if it does not put the computer's king in check. The only problem with the pawn arises when it is about to check — it has two possible moves.

The bishop and the rook are roughly the same. They created no problems and took very little time to develop. For a rook, we add one to the file or row depending on which direction it is moving. The bishop is slightly more difficult as it has to add or subtract one from the file and row — again depending on which way it was moving. When we have found our new position, and before we make it on the board, we check for a number of things.

Complicated pieces

First, we see whether it has reached the other side, or whether it has reached a piece of its own colour. In either case, we know that this is as far as it can go. If it reaches an opponent's piece, it can replace it with its own colour and subtract one from the number of pieces the opponent has on the board. We must also check for a check or checkmate and if there is one, we must either move it to another square or not move that piece at all.

The knight, although it may seem a complicated piece, is very similar to the rook and bishop. You add two to one direction and take one from the other, or any other combination, and you must remember that this piece only moves once, unlike the rook. There are in fact eight moves a knight can make and you must check for the piece going off the board.

The king can move in any direction but only one square at a time, so you can see how easy it is to cater for it in the program. We must remember, however, that it is the piece which must not be attacked. We have to find out whether it is in check both at the beginning, and at the end.

The way this is done is to search along every diagonal, file and row until we meet an edge or a piece of the same colour. If this happens, move on to the next path because the king is not being attacked in that direction. A simple For-Next loop will deal with that.

However, it we find an opponent's piece we must discover which piece it is since all pieces, apart from the queen, are limited in their directions of attack. If the piece is not an attacker, we can consider knight moves away from the king to see if they contain a knight of the opposite colour. If we do not find check then we can move on.

Element number	Value	Comments
1	0	piece either not present or has been tried
2	540	a pawn on its starting square
3	600	a queen being attacked
4	539	a pawn being attacked
5	450	a bishop in the middle, not being attacked
6	460	a castle on its starting square
7	440	a pawn
8	440	a pawn
9	443	a pawn — a random number is added to make sure each game is different
10	510	a knight in a poor position
11	580	a knight being attacked
12	340	the king
13	0	a piece already tried
14	508	a bishop in a poor position
15	570	a pawn on the square before it is promoted
16	0	a piece already tried

Table 1. Points table with list of pieces.

If we find check, we have the other problem of discovering whether it is checkmate. The three ways of eluding check are: moving the king, taking the attacking piece, or blocking. The first is the easiest — move the king to all the possible positions and verify to see if the king is in check. If he is not in check, the situation has been solved.

The next option — taking the attacker — is a far more difficult problem to solve. We must use the same routine to see if any piece is attacking. If it is, we can move it to take the attacker and, as long as we do not cause another check or double check, we have again solved the situation.

Blocking a check with a piece is the most difficult aspect. We can first see if the attacker is a knight, because if it is we cannot block. If it is not, we must, using a For-Next loop, go from every square between the king and the attacker and see if any one of those squares is attacked by one of its own pieces. If it is, we can move the piece to block the attacker.

This again has two conditions, the first is that it is not a double attack, and the second that it does not create a discovered check. If you still cannot find a way out, the computer is in checkmate, and you should have it say so.

The part which decides how well a computer plays is its points table. Each program has its own points' table and a different way of filling it.

The points table is made up of 16 elements, and each element is initially given 500 points. The computer then goes through the board until it finds one of its pieces; it then stops and evaluates the position. It does this by looking at which piece it is and subtracts points if it does not want to move it.

Skill of the game

For example, it is much safer to move a bishop than a queen. The program considers other aspects such as whether it is being attacked. If it is, it will have to move the piece concerned so some points will need adding. Then it can look at its position; if it is in the centre, points will be deducted so that it is less likely to move. Other considerations can be evaluated, depending on how good a game you wish to play, and how much space you have. When this points table is built up, the element with the most points will be tested to see if it can be moved. If it cannot then a zero will be placed in it so that it will not be tested again.

The element with the most points moves first. The skill and standard of the game depends so much on this part of the game that to make it play better, a great deal of work on this section is needed. The results we obtain from a points table need not be accurate enough for playing against some players, but it can be a match for average players if it is dealt with correctly.

The player section is the part which will interact with the player so that he can enter his moves and can see the board. My program displays a full board with the pieces represented as letters:
King-K, Queen-Q, Rook-C, Bishop-B, Knight-N, and Pawn-P.

This is satisfactory for a ZX-80 but if your computer has graphics, or else a user-definable character set, then use the letters. The normal way for entering chess programs into a computer such as a Chess Challenger is by algebraic methods. That is used here — it is also the chess standard.

The final section of the game is the back-up which is just a collection of routines which will deal with such things as:

- Set up points table
- Zeroing of moves already tried
- Loading and using machine code if any is present.
- Reprinting board.
- Putting pieces on the board.
- Different board set-ups — black or white.
- Other tasks which can be used by all of the routines already mentioned. ■