THE GO GAME

The GO game was originated in China. Earliest references to it date back to the Han Dynasty about the time of Christ. It has been the national pastime since. In the Tang Dynasty about the 8th century, it migrated to Japan with many other cultural influences. While the Chinese enjoyed GO purely as a game, the Japanese turned GO into an industry. Currently, Japan boasts the best organized GO societies and maintains the largest group of professional GO players supported by mass population. Western observers would naturally mistake GO as a Japanese game. Being a Chinese, I have to clarify the origin of this game out of my racial pride.

This game is played on a board with a 19x19 grid. White and black stones are played alternately by two players on the cross-points of the grid. Once a stone is put on the board, it may not be moved. The aim is to connect the stones to form territories. The side occupying more territory is the winner. The rules are very simple. Since the stones are static, i.e., they may not be moved after being played also makes it attractive for computerization. The problem is the huge size of the board, which allows an almost infinite number of variations.

The problem in computerizing the GO game must be attacked from many different angles. The coding and storing of games to form databases to be used as guides in playing, or as references for analysis is one aspect. The strategy and the methodology in playing the game is another. One can also separate the beginning and ending of the game from the main game so that the local effects and tactics can be defined more precisely.

I am not capable of solving any of these problems. I can only offer some suggestions and some tools to treat a few very well defined tasks. In this section, I included four programs. The first one shows how to code a complete game and store the game on disk, to be recalled and displayed on a CRT terminal. The second one is the implementation of an algorithm published by Dr. J. K. Millen. The third is a scheme to collect vast numbers of beginning patterns on disk, which might be useful in building a GO playing computer. The last one is a GOMUKO game, which is not related to GO, though played on the same board using the same black and white stones.

CODING AND DECODING OF GO GAMES

I want the computer to read a GO game stored on disc and play it back on a terminal. I want to demonstrate in this program two very unique features of FORTH :

1. Choosing a BASE value best to code and decode GO games;
2. Define a defining word which codes (compiles) a game and also decodes (interprets) the compiled game for play-back.

RULES OF GO GAME

GO game is played on a board of 19 rows and 19 columns. Black stones and white stones are placed alternately on the board at crossing points by two players.

Connecting stones of the same color form territories. The player who manages to surround more territories wins the game.

GO game is better suited for computer processing because the game is static, i. e., stones do not move once they are placed on the board. However, a group of stones may be removed from the board if they are completely surrounded by the opposing stones.

GO GAME COMPILER

A defining word :GAME compiles a game, represented by a list of locations of stones. When a game defined by :GAME is executed, the listed is interpreted and the entire game is played out on a terminal. This is a tool to build a library of games, forming a data base for strategy analysis and possibly a GO computer.

Note the base of location codes is 20, making the codes extreme-compact, comforming to the conventions used by GO societies.

BOARD AND STONES

BLACK, WHITE, CROSS Codes to represent stones and empty points
HOME Return the cursor to upper left corner.
CLEAR Clear the CRT screen and home the cursor.
BLINK Set the cursor mode to 'blink'.
RESET Reset the cursor mode to normal.
GOTOXY Move the cursor to the location specified.
STONE Move the cursor to the board location.
BOARD Print the 19x19 board grids on CRT.
INDEX Print the column numbers.
PLACE Print the stone on the board location.

DISPLAY A GAME

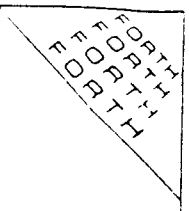
COLOR A flag indicating the color of the stone to be played.
DELAY Delay 5 seconds before playing the next stone.
PLACE Display a stone in blinking mode for 5 seconds and restore it to normal mode.
PLAY Place a stone at the location specified on the stack. If the stone location is greater than 400, delete the stone at the location (modulo 400). Reverse the color for the next stone.
:GAME A defining word to compile a go game from disk block into the dictionary. When the defined game is executed the stones compiled are displayed on the CRT in sequentce.

```
(   BOARD FOR A TEC-70 TERMINAL, CHT, 2-13-81)
HEX
40 CONSTANT BLACK   4F CONSTANT WHITE   2B CONSTANT CROSS
: HOME 1B EMIT 8 EMIT ;
: CLEAR 1B EMIT 5 EMIT ;
: BLINK 1B EMIT 22 EMIT ;
: RESET 1B EMIT 20 EMIT ;
: GOTOXY ( X Y ---) 1B EMIT 0E EMIT 80 SWAP - EMIT
     80 SWAP - EMIT ;
: STONE ( X Y ---)   SWAP 2* 20 + SWAP GOTOXY ;
: BOARD 14 1 DO 14 1 DO I J STONE CROSS EMIT LOOP LOOP ;
: INDEX 14 1 DO I 14 STONE I . 0 I STONE I . LOOP ;
DECIMAL
: PLACE ( STONE LOC ---)   20 /MOD STONE EMIT ;
```

```
(   PLAY, CHT, 2-13-81)
VARIABLE COLOR
: DELAY 5 0 DO 30000 0 DO LOOP LOOP ;
: PLACE  2DUP BLINK PLACE   DELAY   RESET PLACE ;
: PLAY ( N ---)
     DUP 399 > IF   CROSS SWAP 400 - PLACE
     ELSE COLOR @ IF   BLACK SWAP PLACE
                  ELSE WHITE SWAP PLACE
                  THEN COLOR DUP @ NOT !
     THEN ;
: :GAME   CREATE   BEGIN  32 WORD NUMBER DUP , NOT END
     DOES>   BEGIN DUP @   PLAY   2+ DUP @ NOT END ;
EXIT  ( FOLLOWING IS THE FIG-FORTH VERSION OF :GAME)
: :GAME   CREATE BEGIN 32 WORD HERE NUMBER DROP , NOT END
     DOES>   BEGIN DUP @ PLAY 2+ DUP @ NOT END ;
```

```
(   A SAMPLE GAME, CHT, 2-23-81)
:GAME   SAMPLE
GG 33 4G H3 AG 3E FA 64 3C
4C 4B 4D 3A 6G 6F 7F 5F 8G 6D
3H 3G 4H 5G 5H 6H 6I 7E 16H 7H 9E
AE G4 G3 E4 G5 E6 F4 G6 F6 F7
F5 E5 G7 F8 1G6 H6 A6 49 5B GE EG
HG HH HF IH GC EB ED 34 44 45
55 43 54 23 32 25 46 42
52 22 2G 2F 2H 2E CH AI CF DE
DF EF EE BE BF AF CE DD DC CD
BD CC 1BE AE BC AC AB 9C DI 9B EC
BB CB 1AB AA CA HB HA IB IA IG HD
HD CI A8 86 88 94 C3 A3 BE 83
E3 77 68 57 53 132 133 31 73 67 75 78
79 89 8E 8D 9F 9D 99 AD 0
```

GO in Forth



by C. H. Ting

C. H. Ting, Offete Enterprises, 1306 South B Street, San Mateo, California 94402.

Dr. Jonathan K. Millen published a very interesting article on "Programming the Game of Go" in the April issue of *Byte* magazine in 1981. It was a milestone in the computerization of this ancient Chinese board game. There were two major contributions in this work. First was the very simple algorithm for analyzing the progress of the game by counting the liberties of connected groups of stones. The liberty count became the priority scale that could be used to decide the next move by the computer. Second was the demonstration that, using this approach, a very small microcomputer could be programmed to play the Go game, which had always been considered to be too complicated for small computers.

This simple algorithm cannot cope very satisfactorily with all the intricacies of Go; however, it does provide a very firm foundation for future improvements and embellishments. The fact that Millen's program is capable of playing legitimate games, with only minor exceptions, demonstrates well that his algorithm captured the most essential features of the game. It forms a backbone of advanced Go programs in which other features can be added readily to deal with tactics and strategy.

I took Dr. Millen's article as a challenge to a programmer's skill to implement it on a small computer. His implementation was on a KIM-1 with 1 Kbyte of memory. The response time for a move was about one second. These features are hard to match. It would be interesting to see if it could be put on other computers, preferably in a high-level language to make the program transportable. Forth was chosen because of its compactness and efficiency in both programming and runtime. This game was programmed and used as both a practical example and a teaching example of the Forth language in late 1981, when I was back in Taiwan trying to preach the gospel of Forth. It was very effective because the difficulty in this game was well understood over there. A working Go game, in spite of its shortcomings, was impressive.

The Algorithm

The algorithm to play the game of Go as published by Dr. Millen is shown in Figure 1 (page 55). For a more detailed description of the mechanism and rationale in this program, the interested reader should consult the original article. However, the structured English he used to express the algorithm is not difficult to comprehend.

Implementation

The program was first developed on an LSI-11 microcomputer with Poly-Forth as the operating system. The source code spans 8 Kbytes, though loosely packed. The compiled code, including 362 bytes to hold the complete 19-by-19 board, occupied about 3 Kbytes. The response time for a move is about seven seconds.

Later, after I had finished a Forth-79 system for the Apple II and had succeeded in putting Forth in PROMs (replacing the Applesoft BASIC chip set), I tried to transfer the Go game into the Apple. Since the source code is less than 8 Kbytes, it was burned into four 2716's and moved into the Apple on an Apple ROM card. The Forth system in the Apple compiles the source code in ROM to object code stored in RAM and executes the Go game. The interesting thing about this scheme is that I can run a number of games with the same Apple, with all the games on ROM cards in source form.

Transportability

To make sure that the program can be transported to other microcomputers, it is important to avoid memory-mapped graphic representation of the Go board. Standard terminal I/O commands are used to draw the board with stones on the CRT screen. The only terminal-dependent commands are HOME and EOL (HOME returns the cursor to the upper left corner of the screen and EOL erases a line of characters). These two commands manage a stationary board on the screen and facilitate the key-in of the player's moves. They are not necessary if one does not mind the scrolling of the game board.

Although Forth-79 standard was used to present the program, it can be readily modified to fig-Forth or other Forth systems. In fact, I have seen the program running on a couple of CP/M fig-Forth systems, some of which I implemented for local computer manufacturers in Taipei for demonstration purposes.

The Board

Let us turn to the source screens in Listing One (page 63) to see how the algorithm is implemented.

The first two lines in screen 160 declare some constants and variables. BLACK is the code of black stones, having a constant value of 1. Similarly, WHITE has a constant value of 2. COLOR is a variable used to transfer the code of the stone groups currently under processing. LIBERTY is the liberty count (discussed below) accumulated for a group of connected stones; this is the primary indicator of the priority for attention to a given group of stones. MAP is an array of 362 bytes, storing the current board configuration. Each location on the 19-by-19 grid is designated as a byte in this array. A zero in a byte indicates an empty grid location, a 1 indicates a black stone, and a 2 indicates a white stone.

Skipping the rest of screen 160, let us look at the source code in screen 161, because it deals with the display of the game board. HOME moves the cursor to the upper left corner of the screen, preparing to write the new board configuration over the old board. It is desirable to keep an image of the old board on the screen. HOME is written in assembly code, calling the Apple Monitor routine to do the trick.

BK, WT, and CROSS respectively print a black stone, white stone, or grid cross point on the screen. A black stone is the "@" character. A white stone is the capital "O." The grid crossing was first represented by the "+" sign, but later replaced by the period because the "+" signs are too heavy and tend to obscure the stones on the CRT screen.

LIMITS places two numbers, 361 and 0, on the stack because they are often used to designate the loop limits in the analysis of board positions.

INDEX prints a line of indices on the top of the board to designate the columns of the board. BOARD prints the complete board on the screen, with indices on top and left. It takes the contents of the array MAP and prints the appropriate stones on the board. The MAP array is

initialized by CLRMAP, which erases the array to zeros.

PLACE stores a code into one of the array elements. The top of stack has the board position; the next item is the code to be stored in that location. KSTONE removes a stone from the board by storing a zero in the corresponding array element. These two commands change the board configuration. PUT is a utility to put a contiguous block of stones on the board for testing purposes. The color of the stones is specified by the contents of COLOR.

Base 19 Numbering System

GOBASE stores 19 into the user variable BASE, specifying that all subsequent number conversions are to be done in base 19. This is the most appropriate number base for the Go game, since it is played on a 19-by-19 board. The advantage in using base 19 is that all rows and columns can be designated with one character from 0 to 9 and A to J. When typing in a move, the player can type one number – e.g., 35, AF, 9J, etc. – to represent a board location. This number is the exact offset to pick up the proper array element in MAP, without any complicated conversions. The freedom in choosing the best number base for a specific application is a luxury only a Forth programmer can enjoy.

A very good example of how to use GOBASE is the command HANDICAP, which places nine handicap stones on the board to give the computer a fighting chance to compete with a human player. After switching the base to 19 via GOBASE, the board positions can be conveniently specified as 33, 3A, 3F, etc., and PLACEd on the board in a short loop. After HANDICAP, the base is returned to decimal for normal processing.

Liberty Counts

The most important procedure in Millen's algorithm is that which scans a group of connected stones of the same color, and counts the liberties, or the

Listing 1: *Structured English specifications of COUNT module to find and count the liberties of a connected group containing a stone at point "x" of color "color." COUNT calls itself recursively, saving x on the push-down stack during each call.*

```
COUNT(x,color):
  IF x is not off the edge
  THEN
    IF there is a stone at x AND
      it is the given color AND
      it is not marked
    THEN
       mark it
       CALL COUNT(NORTH(x),color)
       CALL COUNT(EAST(x),color)
       CALL COUNT(SOUTH(x),color)
       CALL COUNT(WEST(x),color)
    ELSE IF there is no stone at x
    THEN
       mark the point as a liberty
       increment the liberty count
    END
END
```

Listing 2: *Module specification for the main loop of the Go-playing program and two of its called modules.*

```
MAIN:
  place black handicap stones
  LOOP
    display the board
    get white's move from keyboard
    CALL WEFFECT for the effect of white's move
    CALL BEFFECT to obtain a tentative black move
    CALL PATS to check for a pattern match
    place black stone
END

WEFFECT:
  FOR each point x with a black stone DO
    CALL COUNT(x,black)
    IF the group has no liberties
    THEN remove its stones
    ELSE IF the group has at least one liberty
    THEN
      choose a liberty not on edge line
      IF the group has 1 or 2 liberties
      THEN CALL EVAL for the chosen liberty
    END
END

BEFFECT:
  FOR each point x with a white stone DO
    CALL COUNT(x,white)
    IF the group has exactly 1 liberty
    THEN
      designate it as the black move
      remove the white stones
      EXIT
    ELSE IF the group has 2 or more liberties
    THEN
      choose a liberty
      CALL EVAL for the chosen liberty
    END
END
```

Listing 3: *Module specifications for move evaluation, look-ahead, and pattern matching.*

```
EVAL(move,liberties):
  GLOBAL (best-move, best-liberties)
  IF liberties ≤ best-liberties AND
    LOOKAHEAD(move) ≥ 2
  THEN
     best-move = move
     best-liberties = liberties
  END

LOOKAHEAD(move):
  place black stone at move
  CALL COUNT(move,black)
  remove black stone
  RETURN count of liberties

PATS:
  FOR each white stone DO
    IF there is a pattern in the table
      centered on that white stone
    THEN
      get suggested black move y
      CALL EVAL(y,2)
      EXIT
  END
END
```

Figure 1.

From "Programming in the Game of Go" by Jonathan Millen, originally published in the April 1981 issue of *Byte* magazine. Used with the written permission of Jonathan K. Millen.

open grid positions, around this group. This procedure is implemented in screen 160.

Let us follow the order in which commands are defined in screen 160. ?OUT takes the number at the top of the stack and checks its range. If the number is between 0 and 360 inclusive, a false flag is returned; otherwise, a true is returned. It will be used to abort a counting procedure if the stone location is outside of the board.

?STONE merely returns the code of the stone whose location is given on the top of the stack. The top of stack number is used as an offset into the MAP array to pick up the code stored in the specified location.

NORTH, SOUTH, EAST, and WEST are the commands that convert the current stone location on top of the stack to the neighboring location in the direction indicated by the names of the commands. The returned location might be outside of the board, as NORTH and SOUTH would do if the given location is at the edge of the board. In the cases of EAST and WEST, there is the problem of horizontal wrap-around. If the current location is at the left edge, WEST will then return the number 1000, which is definitely outside of the board. EAST will do likewise at the right edge.

MARK sets the most significant bit of the code whose location is on the top of the stack. This command puts a mark on the locations that were scanned at least once so that the location will not be repeatedly scanned or repeatedly counted.

RECUR is the famous Fig-Forth command allowing compilation of recursive procedures. (Other versions of Forth may use other words, such as MYSELF or RECURSE.) It compiles the code field address of the procedure, which is still in the process of compilation, allowing it to call itself at runtime.

COUNTS walks through an entire group of connected stones and accumulates its liberties in the variable LIBERTY. The color of the stone group is specified in the variable COLOR. It walks by the left-turn rule of the maze theorem, examining the locations north, east, south, and west of the current location. If that neighboring location has an unmarked stone of the same color, it will jump into this location and call itself to continue the walking. It will increment the LIBERTY count if the neighboring location is empty and unmarked. It marks every location it examines so that locations will not be repeatedly scanned or counted. At the end of this recursive process, all the connected stones of the same color will have been scanned and all the liberties around this group will have been marked and counted.

The recursive procedure is naturally limited in its depth by the sizes of the stacks allocated by the system. In this implementation each level of call uses one cell of the return stack and one cell of the data stack. Since in most Forth systems the return stack has about 256 bytes, COUNTS can process a group of 100 connected stones. This is adequate for most games normally played. The system will crash if the stacks overflow.

Removing Stones

In screen 162, the important commands are DESIGNATE and REMOVE. DESIGNATE is to be used in the case that a group of white stones has only one liberty left. The computer will immediately kill this group by playing a black stone into the liberty. After the counting process, this liberty location is the only location having a code of 128, empty and marked. DESIGNATE simply scans the board, and upon finding this location, plays the killing move.

REMOVE erases an entire connected group of stones, whose color is designated in COLOR. REMOVE is very similar to COUNTS, using the same recursive technique to scan the connected group. It marks the member stones in the group by setting the seventh bit of the code as the kill mark, so that the stone scanned will be deleted at the end of the REMOVE process. KMARK performs this marking task.

After the counting or the removing process, many locations examined are tagged by the counting marks or the kill marks. To continue the processing of other stone groups, these marks must be removed to restore the board map to a clean state. This is accomplished by the command UNMARK. UNMARK scans through the whole board and resets all the bits in the codes according to the reset pattern given on the top of the stack. This way, it can be used to selectively reset either the counting bits, the kill bits, or both at the same time.

Lookahead

This Go program only has the ability to look ahead by one step, which is one of its many weaknesses. However, one-step lookahead does give it some similarity to an amateur player. Screen 163 contains the commands to examine the board configuration one step ahead.

The variables BEST-MOVE, BEST-LIBERTIES, BEST-COUNT, and !COLOR are temporary storage locations used to determine the best move in a given board configuration. The command LOOKAHEAD takes a board location given on the top of the stack, tentatively places a black stone in this location, and counts the liberties around the group of black stones connected to this stone. The liberty count is returned to the stack, and the black stone is removed from this location. A better move for the black would have a larger liberty count. The liberty count is then compared to that in the BEST-LIBERTIES to decide whether BEST-MOVE needs to be updated in the later command.

EVAL does the comparison and updating. It takes two values off the stack; the top value is the maximum liberty count allowed in the current configuration and the second is the move to be examined. EVAL calls LOOKAHEAD to get the liberty count of the proposed move. If the resulting liberty count is greater than the count in BEST-COUNT, and the maximum count given on the stack is not greater than the count in BEST-LIBERTIES, the proposed move should be a better choice than the one indicated in BEST-MOVE as the result of prior analysis. In this case the variables BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES are updated. Otherwise, the best move as determined by prior analysis remains intact.

Going through this evaluation process for all the black groups, the computer will be able to pick up the weakest black group and find the best move to maximize its liberty count; hopefully this will save the black group from being captured by the white stones. In attacking the weakest white group, it also chooses the most secured move so that the attacking stone will not be threatened easily.

Pattern Recognition

The following two screens, 164 and 165, contain commands dealing with some elementary pattern recognition, designed to identify some favorable positions to be considered in the evaluation process. Dr. Millen identified seven patterns in which a designated black move usually improves on the overall black configuration. These seven patterns, including some permutations and reflections, are reduced to twelve 16-bit patterns stored in an array named PATTERN. The coding scheme is illustrated in Figure 2 (page 59). An empty location will be examined by coding the stone distributions in the neighboring 3-by-5 regions to its immediate north, east, south, and west. If any of the four codes matches with one of the twelve patterns, this empty location is assigned a liberty count value of 2, a rather high priority.

?RANGE takes a board location off the stack and returns the code stored in the location. The difference from ?STONE is that if the location is outside of the board, ?RANGE will return a 1, the code for black stone. This is the "ghost stone" asserted by Dr. Millen. The entire board can be considered as being surrounded by black stones to take into account some favorable conditions near the edges of the board.

Given a particular board location,

?N, ?E, ?S, and ?W return the code of its northern, eastern, southern, and western neighbor, respectively. They use ?RANGE to find the codes. If the neighbor is off the board, the code of a black stone will be returned.

+4* is a strange command that generates a 16-bit code according to the rules for the codes being constructed in PATTERN. It takes eight items off the stack and packs the least significant two bits in each item into a 16-bit code, to be left on the stack. These items are the codes of stones immediately around a location, and their order on the stack is according to the sequence specified in Figure 1.

PNORTH is given a specific location on the stack and looks at its northern neighboring location. If this neighbor is empty, PNORTH will walk through its eight neighboring locations, push their codes of stones on the stack, and call +4* to pack these eight codes into a single 16-bit pattern code. However, if the northern neighbor is not empty, PNORTH will exit immediately because this neighboring location is not playable and does not need pattern matching.

PEAST, PSOUTH, and PWEST behave similarly. It seems rather clumsy that four similar words have to be defined to do very similar tasks. Some better tools are needed here to clean up the codes.

MATCH is the command that does the pattern matching. It takes a coded pattern off the stack and compares it with the patterns stored in PATTERN. If the coded pattern matches with one of the stored patterns, a true flag is left on the stack; otherwise, a false flag is left. The big command PATS scans the whole board looking for white stones. Finding a white stone, PATS will use PNORTH, PEAST, PSOUTH, and PWEST to examine its surroundings to see if a stored pattern could be identified. It will abort the loop at the first sight of a matched pattern, assign a priority of 2 to this pattern, and call EVAL to do an evaluation. Since PATS is the last pass in the evaluation process, its finding is preferred over other evaluations of the same priority.

In matching a coded pattern with the stored patterns, I initially used the comparison command = to match the patterns. This turned out to be a very restrictive operation because among the eight neighbors there are two or three considered to be crucial – the others can be ignored. FIX takes the coded pattern and ANDs it with the stored pattern to eliminate the non-crucial bits in the coded pattern. The resulting modified pattern is then compared with the stored pattern to determine a match. However, adding FIX to MATCH tends to make the comparison too liberal. It is probably necessary to define a set of masks to be used together with the patterns in order to be able to precisely identify the desired patterns.

Analysis

Screen 166 hosts the two major analysis routines: WEFFECT, which analyzes the effects of the white stones, and BEFFECT, which analyzes the effects of the black stones. The variable ?STOP and the command STOP (which clears ?STOP) will be used later in the main loop to bypass the pattern recognition process if the computer makes a capture. It is rather cumbersome to transmit this stop flag on the stack. A variable is needed to do the job.

UNMARKS initializes the move analysis routines: WEFFECT, which analyze bits in the MAP array and storing the color of the stones to be processed, given on the stack, into COLOR. EXAMINE takes a board location and calls COUNTS to count the liberties of the group of stones connected to this location; EXAMINE then returns the liberty count on the stack.

WCHO and BCHO are two commands that scan the board for empty but

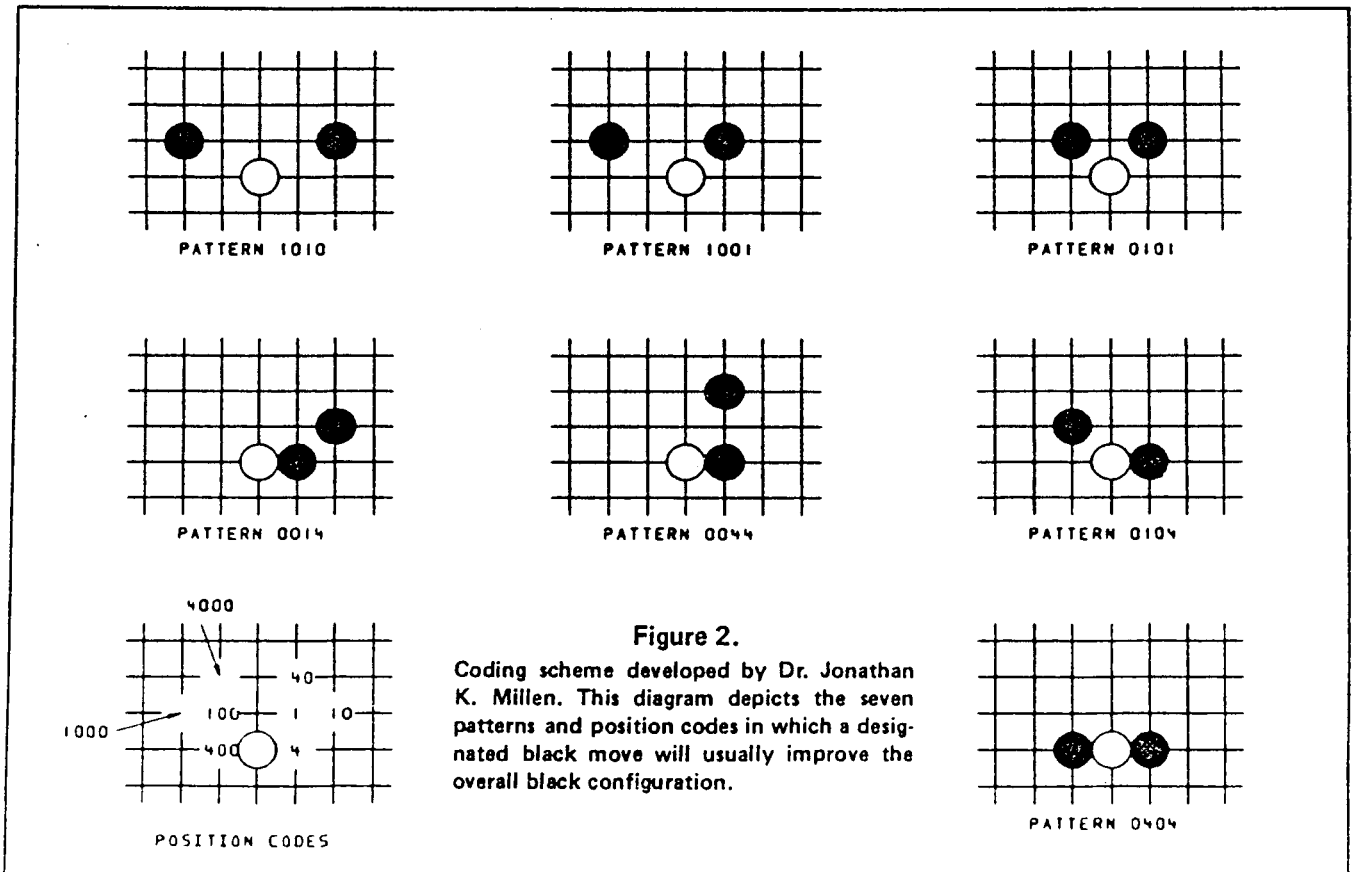


Figure 2.

Coding scheme developed by Dr. Jonathan K. Millen. This diagram depicts the seven patterns and position codes in which a designated black move will usually improve the overall black configuration.

marked locations and push these locations on the stack. These locations are candidates for the next black move, and are to be analyzed using the lookahead technique. A zero is pushed on the stack first as a floor for the location values. These locations will be picked up by CHO to do the actual analysis. CHO scans the locations on the stack and analyzes their priorities. When it finds a zero on the stack, it terminates the analysis. The choice of the best move is stored in BEST-MOVE. WCHO differs from BCHO in that it does not scan the edges of the board, because it is futile for black to play on the board edges to avoid capturing.

?CHO is used to determine whether the liberty count of the current stone group is greater than the value stored in BEST-LIBERTIES. If the liberty count is greater, there is no need to do the move analysis because this move is of a lower priority.

WEFFECT scans all the black stone groups. If a black group has a liberty count of zero, it is completely surrounded by white stones and must be removed from the board. If the liberty count is less than 3 and equal or less than BEST-LIBERTIES, an analysis is performed. If a more pressing move is found, the contents of BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES are updated.

BEFFECT scans all the white stone groups. If a white group has a liberty count of 1, the computer will capture this group by playing into the last liberty. The STOP flag will be cleared at this point. Otherwise, BEFFECT will pick the best attacking move.

Terminal Input

?MOVE in screen 167 is the command that accepts a number from the keyboard as the next move by the human player. It prints a prompting message, clears the rest of the line, and waits for an input string from the keyboard, terminated by a carriage return. It checks the range of the number and also whether the board location is occupied. It will loop until a valid number is entered. Since it uses the standard NUMBER command to do the number conversion, it will abort to the interpreter if an invalid number is entered. This turns out to be a convenient way of stopping the Go game for whatever purpose. The aborted game can be resumed at the point of interruption by the command RESUME.

EOL is the command to clear the rest of the current line. It calls an Apple monitor routine to do the job. It is not absolutely necessary if the player is aware of the way the screen display is managed.

It is important to use numbers in the base 19 format. This way a location is selected by a two-digit number from 00 to JJ, without spaces between the two digits. The columns and the rows of the board are numbered accordingly to facilitate the input process.

The Main Game Loop

The command for playing the Go game is RESUME, which is an infinite loop. In this loop, first the current board and stones are printed out on the CRT screen, the player is asked to input his move, and the computer then goes through the WEFFECT, BEFFECT, and PATS routines to decide its best move. The computer does not know when to stop. It is the human player who usually gets worn out and quits.

.BEST is used to print out the variables stored in BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES. Printing out this information is interesting, as it shows how the computer arrives at its final moves. Since the computer takes about eight seconds to decide its move, printing the intermediate results assures the player that the computer is not idling.

MAIN is the initial entry point of the game. It clears the board, places the nine handicap stones on board, sets the base to 19, and drops into the RESUME loop to play the game. If the game is interrupted by intentionally or unintentionally entering an invalid number, a new game can be started by MAIN or the current game can be continued by RESUME.

Concluding Remarks

The resulting program, compiled into Forth code, occupies about 3 Kbytes of memory. The response time for each move is about eight seconds. These statistics cannot match those of Dr. Millen's implementation on KIM-1 at 1 Kbyte and one second response time. Inefficiency in the high-level codes and poor understandings of the original algorithm are among the most obvious reasons for the decrease in performance. I don't think much can be done in the area of high-level language, since Forth is just about the best high-level language for this type of application. To really make it run faster, many of the routines will have to be coded in assembly. I wonder if anybody will do it, except Dr. Millen.

There are many possible ways of improving the performance of this Forth program. Restricting the scope of searching and analysis will greatly speed up the execution. I tried to limit the searching to two 5-by-5 areas around the black stone last played and the white stone just played. This shortcut speeds up the response time to less than two seconds, without significantly weakening the performance of the computer part of the game. Recording the results of a whole pass of analysis and using them for the next pass will reduce much of the redundant work performed in each pass. The results of analysis need only minor updating and the best move can be quickly selected in the next pass.

There are also some more fundamental problems not properly addressed in the original program, like ko fights, the ladder configuration, Joseki's, etc. These problems can be solved by adding more complicated analysis routines to the framework. The most serious problem is probably that of recognizing two eyes in a connected group of stones and of generating two eyes to build a secured group, immune from being captured. The counting algorithm is not capable of doing this job. To accommodate this capability, a radically different algorithm is needed to do much more sophisticated pattern analysis. A full-scale connectivity analysis could be the best solution. This route is currently being explored.

There will be no end to the lookahead analysis. Professional players look ahead ten or more steps, which is very difficult to simulate by computer. Probably a two- or three-step lookahead in combination with a more extensive pattern analysis process will be sufficient. An important area is the opening games or Joseki's. Large collections of Joseki's compiled into data bases can be useful in guiding the initial games in the four corners on the board. Not much intelligence is needed besides searching the data base for the particular opening. The same is true for the game endings. However, it will then be necessary that the computer have disk storage to hold the data bases.

Computerizing the game of Go is very interesting and challenging. I heartily congratulate Dr. Millen for opening up this field with his excellent work, and hope that many people will also contribute their expertise to a fuller implementation of this game on the microcomputer. Hopefully, we will have the computer Go game reach the level of sophistication of the chess games now available commercially.

DDJ

(Listing begins on page 63)

GO - Listing (Text begins on page 54)

```
160 LIST

 0 ( STONES, CHT,10-1-82)
 1 VARIABLE COLOR VARIABLE LIBERTY 1 CONSTANT BLACK
 2 2 CONSTANT WHITE                 VARIABLE MAP 360 ALLOT
 3 : ?OUT DUP 0< OVER 360 > OR ;    19 CONSTANT 19
 4 : ?STONE ( N --- CODE ) MAP + C@ ;
 5 : NORTH ( N1 --- N2 )  19 - ;    : SOUTH 19 + ;
 6 : EAST  DUP 19 MOD 18 =          IF DROP 1000 ELSE 1+ THEN ;
 7 : WEST  DUP 19 MOD IF 1-         ELSE DROP 1000 THEN ;
 8 ( COUNTS, 10-1-82 )
 9 : MARK MAP + DUP C@ 128 OR SWAP C! ;
10 : RECUR LATEST PFA CFA , ; IMMEDIATE
11 : COUNTS   ?OUT IF DROP EXIT THEN   DUP ?STONE COLOR @ = IF
12            DUP MARK   DUP NORTH RECUR   DUP EAST RECUR
13            DUP SOUTH RECUR   WEST RECUR
14         ELSE DUP ?STONE 0= IF MARK 1 LIBERTY +!
15            ELSE DROP THEN   THEN   ;


161 LIST

 0 ( DISPLAY, CHT, 10-1-82 )  HEX  : BK ."  @" ; : WT ."  0" ;
 1 CODE HOME XSAVE STX, 24 STY, 25 STY, FC22 JSR, XSAVE LDX,
 2          NEXT JMP, END-CODE     DECIMAL
 3 : LIMITS 361 0 ;                : CROSS ."  ." ;
 4 : IND 2 SPACES 19 0 DO I 2 .R LOOP ;
 5 : BOARD   HOME IND  LIMITS DO
 6           I 19 /MOD SWAP 0=      IF 2 .R ELSE DROP THEN
 7           I ?STONE 3 AND DUP 0= IF CROSS DROP
 8           ELSE BLACK = IF BK     ELSE WT THEN THEN LOOP ;
 9 : PLACE ( CODE N --- ) MAP + C! ;
10 : GOBASE 19 BASE !  ;            GOBASE
11 : HANDICAP 33 39 3F 93 99 9F F3 F9 FF   9 0 DO
12         BLACK SWAP PLACE LOOP ;                              DECIMAL
13 : CLRMAP   MAP 362 ERASE ;
14 : KSTONE ( N --- )   MAP + 0 SWAP C! ;
15 : PUT SWAP DO COLOR @ MAP I + C! LOOP ;


162 LIST

 0 ( REMOVE, CHT, 10-22-81)
 1 : KMARK   MAP + DUP C@ 64 OR SWAP C! ;
 2 : REMOVE ( N --- )
 3       ?OUT   IF DROP EXIT THEN
 4       DUP ?STONE 67 AND   COLOR @ = IF
 5          DUP KMARK   DUP NORTH RECUR   DUP EAST RECUR
 6     DUP SOUTH RECUR DUP WEST RECUR KSTONE ELSE DROP THEN ;
 7
 8 ( DESIGNATE, UNMARK, CHT, 10-1-82)
 9
10 : UNMARK ( PATTERN --- )   MAP 361 OVER + SWAP DO
11       I C@ OVER =   IF I C@ 3 AND I C! THEN   LOOP DROP ;
12 : DESIGNATE   LIMITS DO   I ?STONE 128 =
13       IF BLACK I PLACE LEAVE THEN   LOOP ;
14
15


163 LIST

 0 ( LOOKAHEAD, CHT, 10-1-82)
 1 VARIABLE BEST-MOVE   VARIABLE BEST-LIBERTIES
 2 VARIABLE BEST-COUNT      VARIABLE !COLOR
 3 : LOOKAHEAD ( MOVE --- LIBERTY )
 4       BLACK OVER PLACE   COLOR @ !COLOR !   BLACK COLOR !
 5       0 LIBERTY !   128 UNMARK   129 UNMARK
 6       DUP COUNTS   KSTONE   !COLOR @ COLOR !   LIBERTY @ ;
 7
 8 ( EVAL, CHT,10-1-82)
 9
10 : EVAL ( MOVE LIBERTY --- )
11       OVER LOOKAHEAD   >R I   BEST-COUNT @ 1 MAX >
12       OVER BEST-LIBERTIES @ > 0= AND
13       IF BEST-LIBERTIES ! BEST-MOVE !   R> BEST-COUNT !
14       ELSE 2DROP R> DROP THEN ;
15
```

(Continued on page 65)

GO - Listing

(Listing continued, text begins on page 54)

```
164 LIST

 0 ( PATS, MATCH, CHT, 10-1-82)   HEX
 1 VARIABLE PATTERN 14 , 44 , 1400 , 4400 , 1001 , 0110 , 404 ,
 2        401 , 101 , 104 , 1010 ,   1010 PATTERN !   DECIMAL
 3 : ?RANGE ( N --- CODE )   ?OUT IF
 4       DROP BLACK   ELSE ?STONE 3 AND THEN   ;
 5 : ?N NORTH ?RANGE ;              : ?E EAST ?RANGE ;
 6 : ?S SOUTH ?RANGE ;              : ?W WEST ?RANGE ;
 7 : 2* DUP + ;     : +4*   0 8 0  DO DUP + DUP + + LOOP ;
 8 : PNORTH ( MOVE --- PATTERN )   DUP ?N IF DROP 0 EXIT THEN
 9       >R  I NORTH ?E    I ?E    I NORTH EAST DUP ?E
10          SWAP ?N    I NORTH ?W    I
11       ?W    R> NORTH WEST DUP ?W    SWAP   ?N +4* ;
12 : PEAST   DUP ?E IF DROP 0 EXIT THEN
13      >R  I EAST ?S    I ?S    I EAST SOUTH ?S
14         I EAST SOUTH ?E    I EAST ?N    I ?N
15         I EAST NORTH ?N    R> EAST NORTH ?E +4* ;


165 LIST

 0 ( PATS, MATCH, CHT, 10-1-82)  : FIX DUP ROT AND ;
 1 : PSOUTH ( MOVE --- PATTERN ) DUP ?S IF DROP 0 EXIT THEN
 2       >R  I SOUTH ?E I ?E I SOUTH EAST ?E I SOUTH EAST ?S
 3       I SOUTH ?W I ?W I SOUTH WEST ?W R> SOUTH WEST ?S +4* ;
 4 : PWEST   DUP ?W IF DROP 0 EXIT THEN
 5      >R  I WEST ?S I ?S I WEST SOUTH ?S I WEST SOUTH ?W
 6        I WEST ?N I ?N I WEST NORTH ?N R> WEST NORTH ?W +4* ;
 7 : MATCH ( PATTERN--- F)   DUP IF   0 12 0 DO OVER PATTERN I 2*
 8       + @ FIX = IF 1+ LEAVE THEN   LOOP   SWAP DROP THEN ;
 9 : EVPAT    1 BEST-COUNT ! 2 EVAL ;
10 : PATS   LIMITS DO I ?STONE WHITE AND IF   I PNORTH MATCH
11       IF I NORTH EVPAT LEAVE    ELSE I PEAST  MATCH
12       IF I EAST EVPAT  LEAVE    ELSE I PSOUTH MATCH
13       IF I SOUTH EVPAT LEAVE    ELSE I PWEST  MATCH
14       IF I WEST EVPAT LEAVE   THEN THEN THEN THEN THEN LOOP ;
15


166 LIST

 0 ( WEFFECT, BEFFECT, CHT, 10-1-82)
 1 VARIABLE ?STOP   : STOP 0 ?STOP ! ;
 2 : UNMARKS ( COLOR---) 129 UNMARK 130 UNMARK COLOR ! ;
 3 : EXAMINE    128 UNMARK 0 LIBERTY !     COUNTS LIBERTY @ ;
 4 : WCHO 0 341 19 DO I 19 MOD ?DUP IF 18 - IF I ?STONE 128 =
 5       IF I THEN    THEN THEN LOOP ;
 6 : CHO  BEGIN ?DUP WHILE LIBERTY @ EVAL REPEAT ;
 7 : ?CHO BEST-LIBERTIES @ LIBERTY @ < 0= ;
 8 : BCHO 0 LIMITS DO I ?STONE 128 = IF I THEN LOOP ;
 9 : WEFFECT   BLACK UNMARKS   360 BEST-LIBERTIES !
10       LIMITS  DO   I ?STONE BLACK =   IF I EXAMINE
11       0= IF I REMOVE   ELSE LIBERTY @ 3 <
12       IF ?CHO IF WCHO CHO THEN THEN THEN THEN LOOP ;
13 : BEFFECT   WHITE UNMARKS   LIMITS DO   I ?STONE WHITE = IF
14       I EXAMINE 1 =   IF DESIGNATE I REMOVE LEAVE STOP
15       ELSE ?CHO IF BCHO CHO THEN THEN THEN   LOOP ;


167 LIST

 0 ( ?MOVE, CHT, 10-1-82)   HEX
 1 CODE EOL XSAVE STX, FC9C JSR, XSAVE LDX, NEXT JMP, END-CODE
 2 DECIMAL
 3 : ?MOVE BEGIN CR ." YOUR MOVE: "   EOL
 4        0 >IN ! TIB @ 19 EXPECT 32 WORD NUMBER DROP DUP . ?OUT
 5        IF ." RANGE? " DROP 0   ELSE DUP ?STONE 3 AND 0=
 6        IF WHITE SWAP PLACE 1   ELSE ." OCCUPIED." DROP 0  THEN
 7        THEN   UNTIL ;
 8 ( RESUME, MAIN, CHT, 10-1-82 )
 9 : .BEST  3 SPACES BEST-MOVE ? BEST-COUNT ? BEST-LIBERTIES ? ;
10 : RESUME   BEGIN   BOARD ?MOVE   1 ?STOP !   0 BEST-COUNT !
11       WEFFECT .BEST  BEFFECT .BEST    ?STOP @   IF PATS
12       BLACK BEST-MOVE @   .BEST  PLACE THEN   AGAIN ;
13 : MAIN  GOBASE CLRMAP HANDICAP RESUME ;
14 EXIT
15
```

End Listing

180 LIST

```
( COUNTS, CHT,10-22-81)        EMPTY
VARIABLE COLOR VARIABLE LIBERTY 1 CONSTANT BLACK
2 CONSTANT WHITE    3 CONSTANT 3    VARIABLE MAP 360 ALLOT
: ?OUT DUP 0< OVER 360 > OR ;    19 CONSTANT 19
: ?STONE ( N --- CODE ) MAP + C@ ;
: NORTH ( N1 --- N2 )  19 - ;    : SOUTH 19 + ;
: EAST  DUP 19 MOD 18 = IF DROP 1000 ELSE 1+ THEN ;
: WEST  DUP 19 MOD IF 1- ELSE DROP 1000 THEN ;
: MARK MAP + DUP C@ 128 OR SWAP C! ;      : RECUR ;
: COUNTS    ?OUT IF DROP EXIT THEN    DUP ?STONE COLOR @ = IF
            DUP MARK    DUP NORTH RECUR    DUP EAST RECUR
            DUP SOUTH RECUR    WEST RECUR
         ELSE DUP ?STONE 0= IF MARK 1 LIBERTY +!
            ELSE DROP THEN    THEN    ;
' COUNTS 2 -   ' RECUR 2 -  OVER : RECURSE HERE SWAP DO I @ OVER
 = IF OVER I ! THEN 2 +LOOP 2DROP ;    RECURSE
```

181 LIST

```
( DISPLAY, CHT, 10-22-81 )
HEX    MSG BK 2002 , 40 ,    MSG WT 2002 , 4F ,
MSG CROSS 2002 , 2E ,  MSG HOME 1B04 , 2059 , 20 ,  DECIMAL
: LIMITS 361 0 ;
: INDEX 2 SPACES 19 0 DO I 2 U.R LOOP ;
: BOARD    HOME INDEX  LIMITS DO
           I 19 /MOD SWAP 0= IF CR 2 U.R ELSE DROP THEN
           I ?STONE 3 AND DUP 0= IF CROSS DROP
           ELSE BLACK = IF BK ELSE WT THEN THEN LOOP ;
: PLACE ( CODE N --- ) MAP + C! ;
: GOBASE 19 BASE !  ;  GOBASE
: HANDICAP 33 39 3F 93 99 9F F3 F9 FF    9 0 DO
        BLACK SWAP PLACE LOOP ;                              DECIMAL
: CLRMAP    MAP 362 ERASE ;
: KSTONE ( N --- )    MAP + 0 SWAP C! ;
: PUT SWAP DO COLOR @ MAP I + C! LOOP ;
```

182 LIST

```
( REMOVE, CHT, 10-22-81)
: KMARK    MAP + DUP C@ 64 OR SWAP C! ;
: REMOVE ( N --- )
      ?OUT    IF DROP EXIT THEN
      DUP ?STONE 67 AND    COLOR @ = IF
         DUP KMARK    DUP NORTH COUNT    DUP EAST COUNT
    DUP SOUTH COUNT DUP WEST COUNT KSTONE ELSE DROP THEN ;
' REMOVE 2 -    ' COUNT 2 -    OVER RECURSE


: UNMARK ( PATTERN --- )    MAP 361 OVER + SWAP DO
      I C@ OVER =    IF I C@ 3 AND I C! THEN    LOOP DROP ;
: DESIGNATE    LIMITS DO    I ?STONE 128 =
      IF BLACK I PLACE LEAVE THEN    LOOP ;
```

183 LIST

```
( LOOKAHEAD, EVAL, CHT, 10-22-81)
VARIABLE BEST-MOVE   VARIABLE BEST-LIBERTIES
VARIABLE BEST-COUNT      VARIABLE !COLOR
: LOOKAHEAD ( MOVE --- LIBERTY )
      BLACK OVER PLACE   COLOR @ !COLOR !   BLACK COLOR !
      0 LIBERTY !   128 UNMARK   129 UNMARK
      DUP COUNTS   KSTONE   !COLOR @ COLOR !   LIBERTY @ ;
: EVAL ( MOVE LIBERTY --- )
      OVER LOOKAHEAD   >R I   BEST-COUNT @ 1 MAX >
      OVER BEST-LIBERTIES @ > NOT AND
      IF BEST-LIBERTIES ! BEST-MOVE !   R> BEST-COUNT !
      ELSE 2DROP R> DROP THEN ;
```

184 LIST

```
( PATS, MATCH, CHT, 10-22-81)   HEX
VARIABLE PATTERN 14 , 44 , 1400 , 4400 , 1001 , 0110 , 404 ,
      401 , 101 , 104 , 1010 ,   1010 PATTERN !   DECIMAL
: ?RANGE ( N --- CODE )   ?OUT IF
      DROP BLACK   ELSE ?STONE 3 AND THEN   ;
: ?N NORTH ?RANGE ;       : ?E EAST ?RANGE ;
: ?S SOUTH ?RANGE ;       : ?W WEST ?RANGE ;
: +4*   0 8 0 DO 2* 2* + LOOP ;
: PNORTH ( MOVE --- PATTERN ) DUP ?N IF DROP 0 EXIT THEN
      >R  I NORTH ?E    I ?E    I NORTH EAST ?E
         I NORTH EAST ?N    I NORTH ?W    I
      ?W    I NORTH WEST ?W    R> NORTH WEST   ?N +4* ;
: PEAST   DUP ?E IF DROP 0 EXIT THEN
     >R  I EAST ?S    I ?S    I EAST SOUTH ?S
        I EAST SOUTH ?E    I EAST ?N    I ?N
        I EAST NORTH ?N    R> EAST NORTH ?E +4* ;
```

185 LIST

```
( PATS, MATCH, CHT, 10-22-81)
: PSOUTH ( MOVE --- PATTERN ) DUP ?S IF DROP 0 EXIT THEN
      >R  I SOUTH ?E I ?E I SOUTH EAST ?E I SOUTH EAST ?S
      I SOUTH ?W I ?W I SOUTH WEST ?W R> SOUTH WEST ?S +4* ;
: PWEST   DUP ?W IF DROP 0 EXIT THEN
     >R  I WEST ?S I ?S I WEST SOUTH ?S I WEST SOUTH ?W
        I WEST ?N I ?N I WEST NORTH ?N R> WEST NORTH ?W +4* ;
: MATCH ( PATTERN--- F)   DUP IF   0 12 0 DO OVER PATTERN I 2*
      + @ = IF 1+ LEAVE THEN   LOOP   SWAP DROP THEN ;
: EVPAT    1 BEST-COUNT ! 2 EVAL ;
: PATS   LIMITS DO I ?STONE WHITE AND IF   I PNORTH MATCH
      IF I NORTH EVPAT LEAVE    ELSE I PEAST  MATCH
      IF I EAST EVPAT  LEAVE    ELSE I PSOUTH MATCH
      IF I SOUTH EVPAT LEAVE    ELSE I PWEST  MATCH
      IF I WEST EVPAT LEAVE   THEN THEN THEN THEN THEN LOOP ;
```

```
  186 LIST

( WEFFECT, BEFFECT, CHT, 10-30-81)
VARIABLE ?STOP    : STOP 0 ?STOP ! ;
: UNMARKS ( COLOR---) 129 UNMARK 130 UNMARK COLOR ! ;
: EXAMINE    128 UNMARK 0 LIBERTY !     COUNTS LIBERTY @ ;
: WCHO 0 341 19 DO I 19 MOD ?DUP IF 18 - IF I ?STONE 128 =
      IF I THEN    THEN THEN LOOP ;
: CHO  BEGIN ?DUP IF LIBERTY @ EVAL AGAIN ;
: ?CHO BEST-LIBERTIES @ LIBERTY @ < NOT ;
: BCHO 0 LIMITS DO I ?STONE 128 = IF I THEN LOOP ;
: WEFFECT   BLACK UNMARKS   360 BEST-LIBERTIES !
      LIMITS  DO   I ?STONE BLACK =   IF I EXAMINE
      0= IF I REMOVE   ELSE LIBERTY @ 3 <
      IF ?CHO IF WCHO CHO THEN THEN THEN THEN LOOP ;
: BEFFECT   WHITE UNMARKS   LIMITS DO   I ?STONE WHITE = IF
      I EXAMINE 1 =   IF DESIGNATE I REMOVE LEAVE STOP
      ELSE ?CHO IF BCHO CHO THEN THEN THEN   LOOP ;


  187 LIST

( ?MOVE, RESUME, MAIN, CHT, 10-29-81) EMPTY
180 LOAD 181 LOAD 182 LOAD 183 LOAD 184 LOAD 185 LOAD 186 LOAD
: EOL   27 EMIT 75 EMIT ;
: ?MOVE BEGIN CR ." YOUR MOVE: "   EOL
      0 >IN ! S0 @ 80 EXPECT
      32 WORD NUMBER  DUP .   ?OUT
      IF ." RANGE? " DROP 0   ELSE DUP ?STONE 3 AND 0=
      IF WHITE SWAP PLACE 1   ELSE ." OCCUPIED." DROP 0  THEN
      THEN   END ;
: .BEST  CR BEST-MOVE ? BEST-COUNT ? BEST-LIBERTIES ? ;
: RESUME   BEGIN   BOARD ?MOVE   1 ?STOP !   0 BEST-COUNT !
      WEFFECT .BEST  BEFFECT .BEST    ?STOP @   IF PATS
      BLACK BEST-MOVE @   .BEST  PLACE THEN   0 END ;
: MAIN  GOBASE CLRMAP HANDICAP PAGE RESUME ;



  188 LIST
```

JOSEKI -- OPENNINGS OF GO GAME

Just like chess, openings of GO are numerous and very extensive collections and analyses had been worked out. I have a 'Small Dictionary of Joseki's', in which 6000 patterns were recorded and analyzed. GO masters seem to be able to remember most of them and can choose the appropriate ones in their tournament games, while the amateurs stumble over them, sometimes very miserably. This has been my experience. Computer can help a player locating a particular joseki. A joseki data base will also be extremely useful in programming the computer to play GO. The pressing need is a good method to code and to retrieve these patterns efficiently.

CODING THE JOSEKI'S

I am presenting here one way to code and store the joseki's. Using this method, it is possible to store all the 6000 patterns in one floppy disk. It is quite easy to search out any one of interest. However, I haven't worked out the best retrieval scheme to be able to offer deep insights at this moment.

A full implementation of tree structures for joseki's is too complicated due to the number of branches at every level. I choose a simple two level structure. The primary level, or the stems, consists of about 10 moves. 10 to 20 branches can grow on the same stem. A joseki pattern then is composed of a stem and an associated branch. Both the stem and the branch are coded as long Ascii strings, in which two consecutive bytes define a move.

JOSEKI CODES STORED ON DISK

Taking advantages of the disk blocks in FORTH, it is rather convenient to store one stem and all its branches in one block. The stem will be placed in line 0 in the block, and the branches are stored afterwards. Strings are separated by one or more blanks.

The advantage of storing joseki's as strings is that the strings are readable and they can be keyed in using a regular FORTH editor. One disadvantage is that one move costs two bytes. Since joseki's are played in corners, it is possible to code one move using only one byte. This data compression technique is outside of the scope here.

30 LIST

335226223223343141214224255455
6445 655666677458
655666675764746353738384727576824546364737483544383917935149777 9
6556666757647463534773
655666675764746353477358
655666576758515071
6556666474637362726167
655666647463735367
655666637372625364438271816184
65566663737464728362858293
6556666251615053634443
655666625161505363434044648257464736 78
65566661573778
655666516261827273718 3
655666675764746353477358
6556515071667576859 7 4546443564
45464435566547366457667574 85

31 LIST

335226223223343141214 2
2453
242554II453547
2425534551507066
2425455335386573
24254553335386362 55517382
242545533538636272517155473756647366677 64454
155325353627463966 73
14532535362746152439

32 LIST

335226223223342431
2536354521548257
25363545215446385663728382 93
253635452146445443534262415130 73
2536354544211617115443121353426255466364617351714 0
25363545442111544312135342625546636461737251631062017465836 6
25351627172882182164
2535361621284654
25353616212846540403052412112010 01

JOSEKI STRING INTERPRETER

A simple interpreter was developed to decode the stem and branch strings and to play the sequence of moves on a CRT terminal. The interpreter is very simple, occupying only one screen. However, it does need some supporting instructions in the GOMUKO games to draw the GO board and to place stones on the board.

SHOW JOSEKI

This screen shows the typical usage of the instruction JOSEKI. It requires a block number and the total number of branches in this block. It will display all the joseki patterns in sequence by reading the stem and a branch, combining them in a string buffer, and decode the string to be played out on the CRT terminal.

JOSEKI

SBUF	The string buffer in which the joseki code is constructed from the stem and a branch of it.
STRING	First copy the stem to the beginning of SBUF, and then append it with the n'th branch.
S@	Given a branch number and a block number, fetch out the joseki code string to SBUF.
S.	Print the contents of SBUF as a string for diagnosis.
DELAY	A 2 second delay loop.
J.	Decode the string in SBUF and play the joseki on CRT.
JOSEKI	Given the block number and the branches on the stack, play all the joseki patterns in sequence on the CRT terminal.

```
 27 LIST
```

```
 28 LIST

( SHOW JOSEKI, CHT, 4-13-82)
30 18 JOSEKI   31 8 JOSEKI   32 8 JOSEKI   33 7 JOSEKI
34 7 JOSEKI
```

```
 29 LIST

( JOSEKI, CHT, 4-13-82 )
: SBUF HERE 128 + ;
: STRING ( N --- )  32 WORD   DUP C@ 1+   SBUF SWAP MOVE
       0 DO 32 WORD DROP LOOP   HERE C@ SBUF C@ +
       HERE DUP 1+ SWAP C@   SBUF DUP C@ 1+ +   SWAP MOVE
      SBUF C! ;
: S@   ( N BLOCK --- )   >IN 2@ >R >R   0 >IN 2!
      STRING   R> R> >IN 2!   ;
: S.  SBUF COUNT TYPE ;
: DELAY 30000 0 DO LOOP ;    : DELAY DELAY DELAY ;
: J.   SBUF 1+ SBUF C@ OVER + SWAP DO   2 HERE C! 32 HERE 3 + C!
       I HERE 1+ 2 MOVE   I 2 AND   IF WHITE ELSE BLACK THEN
       HERE NUMBER PLACE   DELAY 2 +LOOP   ;
: JOSEKI ( BLOCK N --- )   1+ 1 DO   GOBASE CLRMAP BOARD DELAY
      I OVER S@ J. LOOP   DECIMAL ;
```

GOMUKO

Gomuko is a glorified tic-tac-toe game played on a 19x19 GO game board. The first player having 5 of his stones connected in a straight line, vertical,horizontal, or diagonal, wins. The rule is much simpler than that of GO. It is much easier to play. It is one of the most popular games among oriental children.

I use a weighting scheme to evaluate the importance of a specific location. It is based on the number of stones of the same color in a line of five positions about the location to be evaluated. The weights in the N-S, E-W, NE-SW, and NW-SE directions are grouped together to decide the priority of next move.

BOARD AND STONES

COLOR, WHITE, BLACK	Color codes of the stones.
MAP	An array to store the placements of stones.
WEIGHT	A variable to weight the best move.
!COLOR	Temporary storage of the color of the current stone.
?STONE	Return the color of the stone.
?OUT	Return a true flag if the stone is outside the board.
WT, BK, DOT	Print a white, black, or a blank stone.
HOME	Move the cursor to upper left corner.
SHOW	Given the color and position, print a stone on CRT.
>XY	Change the stone position to x,y coordinates.
XY>	Change the stone x,y coordinates to stone position.
XY@	Fetch the stone color from its x,y coordinates.

DISPLAY AND WINDOW

INDEX	Print the column numbers on top of the board.
BOARD	Print the board with stones stored in MAP.
PLACE	Store a color code into a MAP location.
GOBASE	Set the base to 19 for stone position I/O.
CLRMAP	Clear the MAP array to all zero.

WINDOW Calculate the weight function of the 5 stone configuration. Positions of the stones are on the stack with a zero on the top. The weight is equal to the number of stones of the same color in this configuration. If stones of both colors are detected, the weight is zero.

```
  210 LIST
```

```
  211 LIST

( GOMUKO, STONE & BOARD, 12-4-81, CHT)    EMPTY
VARIABLE COLOR    1 CONSTANT BLACK
2 CONSTANT WHITE    3 CONSTANT 3    VARIABLE MAP 360 ALLOT
VARIABLE WEIGHT    VARIABLE !COLOR
: ?STONE ( N --- CODE ) MAP + C@ ;
: ?OUT    ( N --- F )    DUP 0< SWAP 18 > OR ;                    HEX
: WT 20 EMIT 4F EMIT ;    : BK 20 EMIT 40 EMIT ;
: DOT 20 EMIT 2E EMIT ;
: HOME    1B EMIT 59 EMIT 20 EMIT 20 EMIT ;
: SHOW    ( CODE X Y --- )    1B EMIT 59 EMIT    21 + EMIT
     2* 34 + EMIT    ?DUP IF    BLACK =    IF BK ELSE WT THEN
     ELSE DOT THEN ;                                        DECIMAL
: >XY     ( N --- X Y )    19 /MOD ;
CODE XY>    2 S) S ) ADD    S ) ASL    2 S) S ) ADD    S ) ASL
            S ) ASL    S ) ASL    S ) S )+ ADD    NEXT
: XY@     ( X Y --- CODE )    XY> ?STONE ;
```

```
  212 LIST

( DISPLAY AND WINDOW, CHT, 12-4-81 )
: INDEX 20 SPACES 19 0 DO I 2 U.R LOOP ;
: BOARD    HOME INDEX  361 0 DO
           I 19 /MOD SWAP 0= IF CR 20 U.R ELSE DROP THEN
           I ?STONE 3 AND DUP 0= IF DOT DROP
           ELSE BLACK = IF BK ELSE WT THEN THEN LOOP ;
: PLACE ( CODE N --- )    2DUP >XY SHOW    MAP + C! ;
: GOBASE 19 BASE !  ;
: CLRMAP    MAP 362 ERASE ;

: PUT    SWAP DO COLOR @ I PLACE LOOP ;

: WINDOW    ( N1 N2 N3 N4 N5 0 --- WEIGHT )
     0    5 0 DO  ROT ?STONE ?DUP    IF OR SWAP 1+ SWAP
     THEN LOOP    3 = IF DROP 0 THEN ;
```

WEIGHT ANALYSIS

XY>>	From the x,y data, return the map position. If the x,y coordinate is outside the board, return 361.
EVAL	If the configuration of the 5 positions specified on stack has a weight larger than that was in WEIGHT, put the new weight in WEIGHT.
NS	Analyze the weight of the 5 stones in the NS direction from the stone coordinates on stack.
EW	Analyze the EW direction.
STOPPER	Put a 3 in the 361'th MAP position. Illegal code.

NE, NW DIRECTIONS

NE	Analyze the NE direction.
NW	Analyze the NW direction.
?XY	If x,y is within range, push true one top of them. Otherwise, leave only false.
3DUP	Duplicate the topmost three items on the stack.

WEFFECT AND BEFFECT

WHITE-MOVE	Temporary stoarge for the last white stone.
BLACK-MOVE	Temporary storage for the last block stone.
?END	Look around position N. If there are 5 stones in a continuous line, return a true flag to indicate the end of game.
WEFFECT	If the last white stone completes a line of white stones, accept defect.
BEFFECT	If the position on stack completes a line of black stones, the computer wins the game.
ORDER	Temporary storage for the weight factors.
RECORD	Storage of priority codes derived from ORDERed data.
.R	Print the first three items in RECORD, showing the moves of the highest priorities.
LEVEL	A variable holding the dimension of search area.

```
  213 LIST

( NS, EW, NE, NW, CHT, 12-4-81 )
: XY>>   ( X Y --- N, IF OUT OF RANGE, 361 )
     OVER ?OUT OVER ?OUT OR IF 2DROP 361 ELSE XY> THEN ;
: EVAL ( N1 N2 N3 N4 N5 X --- )
     DROP   0 WINDOW   WEIGHT @ MAX WEIGHT !   ;
: NS   ( X Y --- WEIGHT )       0 WEIGHT !
     DUP 1+ 15 MIN   SWAP 4 - 0 MAX   DO   DUP I 5 + I
     DO DUP I XY>> SWAP LOOP   EVAL   LOOP DROP ;
: EW   ( X Y --- WEIGHT )       0 WEIGHT !
     OVER 1+ 15 MIN   ROT 4 - 0 MAX    DO   DUP I 5 + I
     DO I OVER XY>> SWAP LOOP   EVAL   LOOP DROP ;

: STOPPER   3 MAP 361 + C! ;
STOPPER



  214 LIST

( NE, NW, CHT, 12-4-81 )
: NE    ( X Y --- WEIGHT )       0 WEIGHT !
     4 - SWAP   4 + SWAP   5 0 DO   2DUP   5 0 DO
        2DUP XY>>   ROT 1- ROT 1+   LOOP   DROP EVAL
        1+ SWAP 1- SWAP   LOOP 2DROP ;
: NW    ( X Y --- WEIGHT )       0 WEIGHT !
     4 - SWAP   4 - SWAP   5 0 DO   2DUP   5 0 DO
        2DUP XY>>   ROT 1+ ROT 1+   LOOP   DROP EVAL
        1+ SWAP 1+ SWAP   LOOP 2DROP ;

: ?XY ( X Y --- X Y 1, OR 0 IF OUT OF RANGE )
      OVER ?OUT OVER ?OUT OR IF 2DROP 0 ELSE 1 THEN ;
: 3DUP   DUP 2OVER ROT ;



  215 LIST

( WEFFECT, CHT, 4-8-82)
VARIABLE WHITE-MOVE   VARIABLE BLACK-MOVE
: ?END ( N --- )   >XY 2DUP 2DUP 2DUP   NS WEIGHT @ >R
        EW WEIGHT @ >R   NE WEIGHT @ >R   NW WEIGHT @
        R> MAX R> MAX R> MAX    5 =   ;
: WEFFECT   WHITE-MOVE @   ?END
        ABORT" YOU WIN!!!  "   ;
: BEFFECT   ( N ---)   ?END   ABORT" YOU'VE LOST. "   ;
VARIABLE ORDER 6 ALLOT   VARIABLE RECORD 8 ALLOT
: .R  HOME RECORD DUP ? 2+ DUP ? 2+ ? ;

VARIABLE LEVEL   5 LEVEL !
```

PRIORITY

<SWAP	Swap the two numbers stored at the address.
PRIORITY	Order the data stored in ORDER and compress its contents to a 16 bit value representing priority.
EXAM	Evaluate the surroundings of the given position and store the weight values in ORDER.
!PRIORITY	Evaluate the priority value of the given position. If this value is greater than that at the beginning of RECORD, make it the first RECORD and pushes the ones in RECORD down.
PICK	Verify the stone position before calling !PRIORITY.
SCAN	Scan a vertical line of points and update RECORD accordingly.

MAIN GOMUKO LOOP

SCANS	Scan a square region whose dimension is determined by LEVEL and store the moves of the highest priority into RECORD.
?MOVE	Ask user input of the next move. Check range, occupancy, etc., and display the move on board.
RESUME	The main game playing loop.
MAIN	Entry to the game with initiation.

```
  216 LIST

( BEST-MOVE, CHT, 4-8-82)
: <SWAP    ( ADDR --- )    DUP 2@ 2DUP >
        IF SWAP ROT 2!    ELSE 2DROP DROP THEN ;
: PRIORITY (   --- VALUE )
        6 0 DO    6 0 DO    ORDER I + <SWAP    2 +LOOP 2 +LOOP
        0 8 0 DO 5 *    ORDER I + @     + 2 +LOOP ;
: EXAM    ( X Y --- )    2DUP NS WEIGHT @ ORDER !
        2DUP EW WEIGHT @ ORDER 2+ !    2DUP NE WEIGHT @ ORDER
        4 + !    NW WEIGHT @ ORDER 6 + ! ;
: !PRIORITY    ( X Y --- )    2DUP EXAM    PRIORITY DUP RECORD @ >
        IF RECORD ! RECORD 2+ 2! .R    ELSE 2DROP DROP THEN ;
: PICK  2DUP XY>> ?STONE 0= IF !PRIORITY ELSE 2DROP THEN ;
: SCAN ( X Y I ---)  DUP 1+ OVER MINUS DO
        3DUP - SWAP I + SWAP PICK
        3DUP ROT + SWAP I + PICK    3DUP + SWAP I - SWAP PICK
        3DUP MINUS ROT + SWAP I - PICK    LOOP 2DROP DROP ;


  217 LIST

( ?MOVE, RESUME, MAIN, CHT, 10-29-81)
211 LOAD 212 LOAD 213 LOAD 214 LOAD 215 LOAD 216 LOAD
: EOL    27 EMIT 75 EMIT ;
: SCANS ( XY ---)    >XY LEVEL @ 1 DO 2DUP I SCAN LOOP ;
: ?MOVE BEGIN HOME ." YOUR MOVE: "
        0 >IN ! S0 @ 80 EXPECT
        32 WORD NUMBER  DUP .    DUP 0< OVER 360 > OR
        IF ." RANGE? " DROP 0    ELSE DUP ?STONE 3 AND 0=
        IF DUP WHITE-MOVE !
        WHITE SWAP PLACE 1    ELSE ." OCCUPIED." DROP 0  THEN
        THEN    END ;
: RESUME    PAGE BOARD    BEGIN    ?MOVE    WEFFECT
        RECORD 6 ERASE  WHITE-MOVE @ SCANS
        BLACK RECORD 2+ 2@ XY>    DUP BEFFECT    PLACE   0 END ;
: MAIN  GOBASE CLRMAP RESUME ;


  218 LIST
```