TEACHING STRUCTURED PROGRAMMING ATTITUDES, EVEN IN APL, BY EXAMPLE

T. W.-S. Plum and G. M. Weinberg
Human Sciences and Technology
School of Advanced Technology
State University of New York
Binghamton, New York 13901

ABSTRACT

As a programming assignment in a graduate programming course, students were to program an interactive word game, JOTTO. The language used was APL, under constraints of well-structured programming and complete control of the user-machine interaction. In response to complaints that teamwork was an impediment to programming and that it was not possible to write efficient well-structured programs in APL, the instructors undertook to complete the assignment working as a team. The results of the effort were carefully documented, including experiences with program modification, and are presented here, as they were to the class, to illustrate the principles that should be communicated to professional programmers.

The Assignment

The game of JOTTO was chosen for one assignment in our entry-level graduate professional programming course [1] in order to illustrate at least the following principles:

1. Programming for on-line interaction.
2. Structured programming where time and space efficiency would be critical.
3. Differences between human and machine approaches to the same problem.
4. Differences between languages for implementation, since the game had to be implemented in both PL/I and APL, and a comparison report written.
5. Differences between batch (PL/I) and on-line (APL) programming.

On the students' comparison reports, the most frequent complaint was that it was simply not possible to work in a structured way in APL, especially when there were space and time constraints. The problem had been carefully designed to induce space and time problems, for a list of 2688 five-letter words from the Merriam-Webster Pocket Dictionary [2] had to be used for the final tests in a 30K byte workspace, under constraint that the user should not be troubled by the interaction time.

The second most frequent complaint was that when working on-line, as in APL, a team approach hindered progress. Since one of the main objectives of the course was to impart a well-structured team approach to students whose previous experiences were likely to be lone wolf and lacking structure, the two instructors decided that a drastic lesson was needed. Therefore, they undertook to program the problem themselves.

JOTTO is a word-guessing game. Two opponents each select a five-letter word which the opponent must guess by an alternate guessing-and-reply process. Each guess is a five-letter word, and each reply is the number of letters in common between the "hidden" word and the guessed word. (See Appendix 1 for more complete rules, and Appendix 2 for a typical game against the machine.)

The human strategy in JOTTO is generally one of letter-by-letter elimination, but for the computer, with its infallible memory, a simple-minded sieve seems to be the superior approach. The one team out of 18 that attempted a letter-by-letter approach never managed to complete the project--all others used the sieve, in one variation or another.

Since the sieve technique uses all possible information from the guess and reply, the only area for improvement of guessing play lies in second-order strategic consideration--which word to guess on a given turn. Since one of the objectives of the assignment was to see how easily we could make experimental modifications in well-structured programs, our final program makes a modest essay in this area. We guess a word that minimizes a particular estimate of the length of the game; but as luck would have it, our program was beaten by the champion of the class tournament--a program that used no further strategy beyond the simple sieve. Of course, to evaluate such strategies properly, a long series of games would be needed, but this will be the topic of another report.

Initial Implementation

The program was consciously a two-man team effort from the beginning. (The final version of the program is presented in Appendix 3.) One two-hour planning session produced the coding for the top-level function, JOTTO, and a division of its subroutines between the two team members. The mainline was born as a structured flow chart in the style of Nassi and Schneiderman [3]. The question of "Who gets what?" was answered by: (a) examining the interdependence of the routines, influenced in concept, but not in notation, by Parnas [4]; and (b) personal preferences to try out pet ideas. The top-level routine (JOTTO) was never altered after that first session--a sign of good modularization and good luck.

Structured programming was the pedagogic thrust of the program. Modularity, and its payoff for modifiability and comprehendability, was our first concern. A uniform embodiment of DO-loops in APL was still in experimental stages; a uniform representation of IF statements was not even attempted. But we adhered to the convention that the flow structure of each routine, however it was realized by specific statements, would be strictly composed of DO and IF structures. The small size of the modules keeps the small deviations from convention still comprehensible (as in INWORD, REPLY, and TERMINATION).

Modularization was supported by the initial design, but also by the technique of setting a conscious limit of 25 uncluttered APL statements--no "one-liners" --per module. If and when a module grew longer or threatened to grow longer than this in the original paper coding, we took this growth as a sign to modularize further. The careful initial planning and clear program structure prevented any cancerous growth of modules while they were under test at the terminal, one of the most typical student difficulties when working with APL.

Another student problem is encouraged by most APL texts, which seem to set store by the shortest possible names and never bother to localize variables. We set ourselves a standard of localizing all variables except those explicitly needed to be global, and of choosing names as a convenience to the reader, not as a shortcut (ill-advised) for the writer. The reader should judge for himself the extent to which our conventions succeeded, starting with the top-level JOTTO routine and moving down the program structure as guided by the structure diagram shown as Figure 1.

Total working time in addition to the four man-hours of planning was 3.5 man-hours at the terminal for keying the programs and detecting and correcting the 5 keying and 2 logical errors that occurred. Thus, in 7.5 man-hours of effort (and less than 12 hours of elapsed time), a zero-level working version was obtained. As planned in the construction of the assignment, however, this version could not operate for the full 2688-word list in a 30K workspace, so a space-optimization step became necessary. We should note, however, that half of the eighteen student teams did not pass this point, and never obtained a version that would work with the full word list, even though they averaged well over 100 man-hours for their APL version, and took 6 weeks of elapsed time.

Space Refinements

The most frequent objection to small-module construction is overhead inefficiency, yet the modularity of this project yielded insight into opportunities for global optimization, provided the extra programmer time to make the revisions, and allowed the revisions to be made with limited side effects. We shall discuss the revisions in some detail so that the reader can see just how this happened.

In the simplest version of the sieving process, one would simply erase or delete the words which are no longer possible winners. However, this prevents the program from checking the legality of the opponent's guesses, so from the first session we agreed to preserve words. Our choices for embodying the sieve, as we saw them, were to keep a separate list of sieved ("still potential winner") words, or to keep a list of the indices of the words. Storage considerations (five bytes per word, four bytes per index) led us to establish a global variable INDEX which maintained the indices of the sieved words. Indeed, the problem had been designed as a classroom assignment with such considerations in mind, in order to rule out certain blind APL approaches and force the students into at least one "space optimization" and one "time optimization" step. The design was quite successful in this regard, and forced even the instructors to make this modification.

In the zero-level version, the SIEVE subroutine removed from INDEX those indices of WORDS which were eliminated by the most recent guess-and-reply, and INDEX was simply initialized by the APL iota function to have all indices from 1 to the size of WORDS. Since APL modifies the size of variable arrays dynamically, the maximum storage requirement came only at the beginning, when INDEX was at its full size.

We found that by removing the creation of INDEX from the INITIAL routine and creating only the needed indices on the first pass through SIEVE, the space problem was solved--at least statistically--since typically three-fourths of

INDEX is deleted as a result of the very first guess. Because of modularization, only INITIAL, GUESS, and SIEVE needed changing--in obvious ways--and the modification was made in less than one man-hour of work.

Time Refinements

The full word list could now be used, but we found that the program ran agonizingly slowly on its first move--perhaps three to five minutes of elapsed time if the APL system was not too heavily loaded. Again, this was an explicit design criterion that had gone into the problem specification, for we had estimated that any simple-minded approach would be too slow to satisfy a human player, given the inefficiencies of APL's interpretive implementation.

The approach we teach to execution time refinements is based first of all, of course, on choosing appropriate algorithms, but within a particular algorithm, on the detection of the critical point [5]. When, as in this case, we are searching for more than a 50 percent improvement in operating speed, there can be at most one critical point, which in this program clearly had to be COMMONWITH --the routine that calculated the number of letters in common between two words. Version 2 of the COMMONWITH routines was a direct substitution involving a matrix operation instead of a loop, which gave an overall speed increase of close to 50 percent. This is a typical local optimization: no other parts of the program were affected--a sign of appropriate modularization--and less than one man-hour was needed.

At this point, the program played at a more or less acceptable speed, as long as the system was not saturated. This refinement, then, made the program responsive to the conditions of the assignment--something that none of the teams had accomplished (so that our tournament had to be run without regard to execution times being "reasonable"). Thus, by a modular, stepwise approach, we had achieved in less than ten man-hours what none of the teams could do in more than 100.

Even with this change, however, the program was too slow to consider modifications for improving strategic play. Unless considerable speedup could be obtained, improving the quality of play seemed out of the question. However, since the main objectives had been met and a clear demonstration had been made to get the class back on track, the problem was allowed to rest there for several weeks.

During a lull in activity, we brought the problem up again for discussion and noticed the possibility of a neat solution. If not for the problem of duplicate letters, the counting of common letters could have been accomplished by the simple APL expression,

$+/A\epsilon B$

($A\epsilon B$) produces a vector of ones or zeros, with ones for those elements of A that are also elements of B. The sum-reduction (+/) applied to this vector then counts the ones, which, if there are no duplicates, will be the number of letters in common between words A and B. Indeed, three of the student teams had incorrect programs because they thought this function counted all cases correctly.

If an algorithm doesn't have to be correct, it can be as fast as you like. Our problem was to preserve the speed and change the correctness of the approach, which we could do by forcing the words not to have letters in common. With some extra initial work, the program could encode the words so that the second occurrence of a given letter had a different character code from the first, and so forth. The third version of COMMONWITH then became a simple counting of equal letters; in APL, $+/A\epsilon B$. This short code was inserted directly into inner loops in HOWSPLITS and SIEVE, the most frequent calls, but other calls on COMMONWITH were left intact. This change in the major global variable had effects in several functions: COMMONWITH, GUESSWASBAD, HISWORDISBAD, HOWSPLITS, INITIAL, INWORD, REPLYWASBAD, and SIEVE, as well as creating the new functions DECODE, ENCODE, and INITIALENCODE.

In many ways, this was a worst-case modification--changing the fundamental global data structure on which the game was based--and this showed in the effort of making the change. This change was the only one of the refinement steps which introduced an error--we omitted to insert DECODE in line 7 of REPLYWASBAD. This error was not found by "debugging", for in order for it to show up we would have had to use a bad reply consisting of a word with some multiple letter, a rather unlikely case. Instead, the error was found by symmetry arguments when reading a draft of this paper--a neat example of the power of program reading [6], and of the value of having a clear, modular structure that can be read even months after the program was written. (The initial change was made in violation of our usual practice of egoless code reading, as one of the authors was out of the country at the time.)

Altogether, this modification took two man-hours of effort (not counting the "thinking time" to come up with it, or the time to find the bug that we didn't detect, neither of which times we can estimate fairly). The program now ran so fast that it was almost imperceptible to the user, so we felt we had sufficient capacity to make some strategic improvements.

Strategic Refinements

The first ("zero-th" officially) version of GUESS simply picked a random element of INDEX and guessed its associated word. This one-liner allowed quick testing of the entire system, which was all written in the evening of the first planning session. A more intelligent version of GUESS became the official "first" version; this one tried several words from the word list and used the one which performed best on a certain test of splitting the sieved words (in INDEX) into equalish-sized classes.

The time saved by modifying COMMON-WITH gave us enough room to test sufficiently large classes of words to come up with good guesses, and play improved accordingly--by an average of over one move per game. In making these changes, however, we pushed the space requirement up so that once in a while there wasn't sufficient room for INDEX even after the first move, if the guess didn't eliminate enough words--an unintentional but typical "change-makes-change" situation.

In order to cope with this situation, we did what we probably should have chosen to do the first time we encountered a space problem--we changed SIEVE to keep track of sieved words by their position in the WORD list, which was sorted in-place into two classes, possible and eliminated. This change got rid of INDEX altogether, and so affected the two other routines that used it--GUESS and INITIAL. It had the interesting additional advantage that since the WORD list is reordered with every game, the program will definitely play differently each time the same word is presented, thus preventing certain adaptations by the opponent.

On the other hand, shuffling the word list does make the system harder to study if what we are interested in is the JOTTO game itself, and not the programming process. In considering the modification of our system from an interactive toy to a "batch" experimental tool that plays games against itself to study strategy, we discovered one "failure" of our design. This failure would have cost more in modification effort than we might have needed had we followed Parnas' suggestions more closely. In retrospect, we see that all communication with the terminal should have been buried in common GET and PUT subroutines, so that the system could have been modified to off-line simply by modifying these two functions.

In general, we have followed this practice in other interactive systems we have produced [7], but for some reason did not do it here, but rather scattered communication operations throughout the code. While they are rather readily recognized, and we could have converted them for this presentation, we decided to leave them as an example of how any program development could be improved upon. We would certainly recommend, however, that any interactive system be programmed with all communication functions buried a la Parnas and kept at the lowest level possible.

Summary and Conclusion

We have presented our experience with a classroom problem designed to have certain realistic aspects so as to prepare our programmers for the world outside the university. We hope we have demonstrated, as we did to our students, that approaching the programming process in a structured way can yield factors of ten or more improvement in productivity. (Please note that the difference in our results from those of the 18 programmer teams cannot be attributed to mere "experience" or familiarity with the APL language. Several of the teams had total experience equal to or surpassing the team of instructors, and many of the students had far more hours of APL time under their belts.)

We do not present our programs as some sort of optimum "solution"--instructors should avoid doing this even when they believe it to be the case. We have already been critical of our modularization of communication functions, and of some of our stylistic variations. The reader may see strategic or analytic improvements which will result in a significantly better game being played, for the game of JOTTO has a good deal more theoretical interest than appears on the surface.

But this paper is not about JOTTO. It is about the way people write programs, and teach others to write programs. We believe that programming is a practical subject, not a mathematical one, and must be taught by instructors who are prepared to demonstrate how the principles they espouse may be put into action. We believe that "structured programming" does not mean some rigid set of mathematical rules imposed on programmers, but an attitude about programming that says you can always improve if you only examine the way you currently do things. If, through exercises such as these, frankly discussed with our students, we can make them program self-consciously, we shall have succeeded as teachers.

References

[1] Thomas W.-S. Plum and G.M. Weinberg, "Teaching Experienced Professionals: Remedial Programming", in Proceedings of the IFIP Working Conference on Programming Teaching Techniques, North Holland Publishing Co., Amsterdam, 1973.

[2] The New Merriam-Webster Pocket Dictionary, G. & C. Merriam Co., New York. (Available, for scholarly use, on magnetic tape.)

[3] I. Nassi and B. Schneiderman, "Flowchart Techniques for Structured Programming", Technical Report Number 8, SUNY at Stony Brook.

[4] D.L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12. December 1972, pp. 1053-1058.

[5] G.M. Weinberg, PL/I Programming: A Manual of Style, McGraw-Hill, New York, 1970.

[6] G.M. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.

[7] D. Weinberg and G.M. Weinberg, "Experimental Use of a Computer in the Field --Kinship Information", Social Science Information, December 1972.

Appendix 1: Rules of JOTTO

1. The object of the game of JOTTO is to guess your opponent's concealed 5-letter word before he guesses yours.
2. If the two players require the same number of turns, then that game is a draw.
3. On each turn, each player guesses one 5-letter word. If the word has the same five letters as the concealed word (example: PARSE = SPARE = PEARS = SPEAR = PARES = RAPES) the game terminates.
4. If the game terminates, each player is entitled to check the previous responses (see #7) of the opponent against the concealed word. If he detects an incorrect response, the opponent loses. (Both lose if both have been incorrect.)
5. Each concealed word must be chosen from a prespecified vocabulary list. At each move, a player may check the word against the vocabulary. If the word is not in the vocabulary, the player choosing it loses.
6. If neither player has guessed the other's word after 20 moves, the game terminates. Either player may still win under rules 4 or 5, but if neither wins, the game is a draw.
7. On each move, a player is required to indicate how many letters his concealed word has in common with the word guessed by his opponent.

Example 1: Concealed word = SNARK

Guessed word	Response	Letters in common
BLAST	2	A S
GLYPH	0	---
RAINS	4	R A N S
RANKS	5	R A N K S

Example 2: Concealed word = EERIE

Guessed word	Response	Letters in common
BLAST	0	---
QUIET	2	I E
REEKS	3	R E E
LEVEE	3	E E E
EERIE	5	E E R I E

Appendix 2: A Sample Game Against the Computer

```
      JOTTO
IF YOU WISH TO MAKE THE FIRST GUESS
TYPE THE NUMBER 1
ANY OTHER REPLY AND I WILL START
?:
      0

 MOVE NUMBER 1
I PREDICT 6 MORE MOVES
MY GUESS: TABLE
HOW MANY LETTERS DOES MY GUESS SCORE?
3
PLEASE GUESS ANOTHER
SIEVE
1 LETTERS IN COMMON
332 POSSIBILITIES REMAIN

 MOVE NUMBER 2
I PREDICT 4 MORE MOVES
MY GUESS: THOSE
HOW MANY LETTERS DOES MY GUESS SCORE?
1
PLEASE GUESS ANOTHER
BLAST
0 LETTERS IN COMMON
141 POSSIBILITIES REMAIN

 MOVE NUMBER 3
I PREDICT 4 MORE MOVES
MY GUESS: BUYER
HOW MANY LETTERS DOES MY GUESS SCORE?
2
PLEASE GUESS ANOTHER
QUIET
2 LETTERS IN COMMON
44 POSSIBILITIES REMAIN

 MOVE NUMBER 4
I PREDICT 2 MORE MOVES
MY GUESS: LAGER
HOW MANY LETTERS DOES MY GUESS SCORE?
3
PLEASE GUESS ANOTHER
GUESS
1 LETTERS IN COMMON
13 POSSIBILITIES REMAIN

 MOVE NUMBER 5
I PREDICT 2 MORE MOVES
MY GUESS: ULTRA
HOW MANY LETTERS DOES MY GUESS SCORE?
3
PLEASE GUESS ANOTHER
CLEAR
1 LETTERS IN COMMON
4 POSSIBILITIES REMAIN

 MOVE NUMBER 6
I PREDICT 1 MORE MOVES
MY GUESS: LABOR
HOW MANY LETTERS DOES MY GUESS SCORE?
2
PLEASE GUESS ANOTHER
HOVER
1 LETTERS IN COMMON
2 POSSIBILITIES REMAIN

 MOVE NUMBER 7
I PREDICT 1 MORE MOVES
MY GUESS: VALUE
HOW MANY LETTERS DOES MY GUESS SCORE?
5
PLEASE GUESS ANOTHER
COLOR
1 LETTERS IN COMMON
I BELIEVE THE GAME IS OVER
BUT I HAVE TO CHECK THE PLAYS
MY HIDDEN WORD WAS: FICHU
WHAT IS YOUR WORD?
VALUE
I CLAIM VICTORY
TO PLAY AGAIN TYPE:
     JOTTO
```

```
    ∇ JOTTO;IMOVEFIRST
[1]   ⍝VERSION 1: MAIN LINE, PLAYS ONE GAME WITH THE USER
[2]   ⍝ ASSUMES WORD LIST *WORDS* IS IN CORE.
[3]   ⍝ SUBMODULES:INITIAL,GUESS,SIEVE,TERMINATION,ANSWER,REPLY
[4]   ⍝ GLOBAL VARS: MYGUESSES,HISGUESSES,HISREPLIES,INDEX,MOVE
[5]   IMOVEFIRST←INITIAL
[6]   MOVE←0
[7]  NEXT:→FINISH IF 20<MOVE←MOVE+1
[8]   ''
[9]   ' MOVE NUMBER ';MOVE
[10]  →MEFIRST IF IMOVEFIRST
[11] HEFIRST:HISGUESSES[MOVE;]←ANSWER
[12]  MYGUESSES[MOVE;]←GUESS
[13]  HISREPLIES[MOVE]←REPLY
[14]  →TERMTEST
[15] MEFIRST:MYGUESSES[MOVE;]←GUESS
[16]  HISREPLIES[MOVE]←REPLY
[17]  HISGUESSES[MOVE;]←ANSWER
[18]  ⍝
[19] TERMTEST:→FINISH IF EITHERGUESS
[20]  SIEVE
[21]  →NEXT
[22]  ⍝
[23] FINISH:TERMINATION
    ∇

    ∇ IMOVEFIRST←INITIAL
[1]   ⍝VERSION 3: SETS UP 'FAST-SCAN' WORDS, ASSUMES 'IN-PLACE' SIEVE
[2]   ⍝ GLOBAL VARS: HISGUESSES,MYGUESSES,HISREPLIES,MOVE,MYWORD,WORDS,
[3]   ⍝              LASTSIEVED,FISHY,CHUNKSIZE
[4]   ⍝ SUBMODULES: INITIALENCODE
[5]   HISGUESSES←MYGUESSES← 20 5 ⍴'?'
[6]   HISREPLIES←20⍴0
[7]   LASTSIEVED←1↑⍴WORDS
[8]   FISHY←0
[9]   CHUNKSIZE←200
[10]  INITIALENCODE
[11]  ⍝   SET RANDOM SEED WITH 6⌶1,⌶20
[12]  MOVE←6⌶1,⌶20
[13]  MOVE←¯1
[14]  ⍝    CHOOSE WORD
[15]  MYWORD←WORDS[?(⍴WORDS)[1];]
[16]  ⍝     DECIDE ON FIRST MOVE
[17]  'IF YOU WISH TO MAKE THE FIRST GUESS'
[18]  'TYPE THE NUMBER 1'
[19]  'ANY OTHER REPLY AND I WILL START'
[20]  IMOVEFIRST←(1≠1+□,2)
    ∇

    ∇ INITIALENCODE;I;NWORDS
[1]   ⍝VERSION 1: CREATE THE GLOBAL VARS FOR ENCODE AND DECODE
[2]   ⍝ CREATES GLOBAL VARS: ALPHA,ENCODEALPHA,ENCODEMASK
[3]   ALPHA←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[4]   ENCODEALPHA←'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
      ⍺⍳⍵⌊∈_∇Δ⍳⍬''□|⊤⍬×?⍴⌈~+∪⍵⊃+⊂1234567890+×←  <≤=≥>≠∨∧-+→'
[5]   ENCODEMASK← 5 5 ⍴ 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1
       0
[6]   →0 IF 'A'∈WORDS
[7]   I←0
[8]   NWORDS←1↑⍴WORDS
[9]  NEXTA:→QUITA IF NWORDS<I←I+1
[10]  WORDS[I;]←ENCODE WORDS[I;]
[11]  →NEXTA
[12] QUITA:→0
    ∇
```

```
      ∇DECODE[□]∇
    ∇ R←DECODE W
[1]   ⍝VERSION 1: CONVERTS A CODED WORD INTO STANDARD LETTERS
[2]   ⍝ GLOBAL VARS: ALPHA,ENCODEALPHA
[3]   R←ALPHA[1+26|¯1+ENCODEALPHA⍳W]
    ∇

      ∇ENCODE[□]∇
    ∇ R←ENCODE W
[1]   ⍝VERSION 1: ENCODES WORD INTO 'FAST-SCAN' CODE
[2]   ⍝ GLOBAL VARS:ALPHA,ENCODEALPHA,ENCODEMASK
[3]   R←ENCODEALPHA[(ALPHA⍳W)+26×+/ENCODEMASK∧W∘.=W]
    ∇

      ∇COMMONWITH[□]∇
    ∇ R←A COMMONWITH B
[1]   ⍝VERSION 3: USING WORDS PROCESSED BY *ENCODE*, JUST COUNT EQUAL L
      ETTERS
[2]   R←+/A∈B
    ∇

      ∇ANSWER[□]∇
    ∇ HISTRY←ANSWER
[1]   ⍝VERSION 1: ACCEPTS WORD FROM USER, RETURNS SCORE (0 THRU 5)
[2]   ⍝ SUBMODULES: INWORD,COMMONWITH
[3]   HISTRY←INWORD 'PLEASE GUESS ANOTHER'
[4]   MYWORD COMMONWITH HISTRY;' LETTERS IN COMMON'
    ∇

      ∇REPLY[□]∇
    ∇ N←REPLY
[1]   'HOW MANY LETTERS DOES MY GUESS SCORE?'
[2]   GET:→ERR IF 1≠⍴⍴N←⍞
[3]   →ERR IF 5<N←¯1+'012345'⍳N
[4]   →0
[5]   ERR:'WHAT? A NUMBER FROM 0 TO 5 PLEASE.'
[6]   →GET
[7]   ⍝  REPLY ACCEPTS INPUT; RETURNS N∈[0,5]
    ∇

      ∇EITHERGUESS[□]∇
    ∇ RETURN←EITHERGUESS
[1]   ⍝YES IF EITHER HAS GUESSED 5 LETTERS
[2]   RETURN←1
[3]   →(HISREPLIES[MOVE]=5)/0
[4]   →((MYWORD COMMONWITH HISGUESSES[MOVE;])=5)/0
[5]   RETURN←0
    ∇

      ∇INWORD[□]∇
    ∇ R←INWORD MESSAGE
[1]   ⍝VERSION 2: ENCODE THE INPUT INTO 'FAST-SCAN' CODE
[2]   ⍝ SUBMODULE: ENCODE
[3]   MESSAGE
[4]   GET:→ERR IF 5≠⍴⍴R←⍞
[5]   →ERR IF~∧/R∈'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[6]   R←ENCODE R
[7]   →0
[8]   ERR:'EH?  5 LETTERS, PLEASE.'
[9]   →GET
    ∇
```

```
      ∇SIEVE[□]∇
    ∇ SIEVE;N;MINE;U;T
[1]   ⍝ VERSION 2: SIEVES IN-PLACE, USING 'FAST-SCAN' WORDS
[2]   ⍝  GLOBAL VARS: HISREPLIES,MOVE,FISHY,MYGUESSES,LASTSIEVED,WORDS
[3]   →0 IF FISHY
[4]   N←HISREPLIES[MOVE]
[5]   MINE←MYGUESSES[MOVE;]
[6]   U←0
[7]   T←LASTSIEVED
[8]  NEXTC:→QUITC IF U≥T
[9]  NEXTA:→NEXTA IF N=+/MINE∈WORDS[U←U+1;]
[10]  LASTSIEVED←U-1
[11]  T←T+1
[12] NEXTB:→NEXTB IF(1<T)∧N≠+/MINE∈WORDS[T←T-1;]
[13]  →QUITC IF LASTSIEVED≥T
[14]  WORDS[U,T;]←WORDS[T,U;]
[15]  LASTSIEVED←U
[16]  →NEXTC
[17] QUITC:→ERROR IF N≠+/MINE∈WORDS[1;]
[18]  LASTSIEVED;' POSSIBILITIES REMAIN'
[19]  →0
[20] ERROR:FISHY←1
[21]  ' THERE IS SOMETHING FISHY ABOUT YOUR ANSWERS'
[22]  ' BUT I WILL CONTINUE TO PLAY UNTIL ONE OF US GETS 5 CORRECT'
[23]  LASTSIEVED←1
    ∇

      ∇GUESS[□]∇
    ∇ R←GUESS;N;JSIEVED;NWORDS;M;MAXCLASS;IUNSIEVED;T;NPOSS;I
[1]   ⍝ VERSION 3: ASSUMES 'IN-PLACE' SIEVE, AND 'FAST-SCAN' WORDS
[2]   ⍝ SUBMODULES: HOWSPLITS
[3]   ⍝ GLOBAL VARS: WORDS,LASTSIEVED,CHUNKSIZE
[4]   NWORDS←1↑⍴WORDS
[5]   NPOSS←LASTSIEVED
[6]   N←LASTSIEVED⌊CHUNKSIZE
[7]   JSIEVED←N?LASTSIEVED
[8]   M←60⌊NWORDS
[9]   MAXCLASS←M⍴999
[10]  IUNSIEVED←M?NWORDS
[11]  IUNSIEVED[⍳I]←JSIEVED[⍳I←N⌊⌊M×12÷N]
[12]  I←0
[13] NEXTA:→QUITA IF M<I←I+1
[14]  MAXCLASS[I]←+/(1+4●T×NPOSS)×T←(÷N)×(T≠0)/T←5↑HOWSPLITS WORDS[
      IUNSIEVED[I];]
[15]  →NEXTA
[16] QUITA:I←IUNSIEVED[1↑⍋MAXCLASS]
[17]  R←WORDS[I;]
[18]  DIGITS 3
[19]  'I PREDICT ';⌊0.5+⌊/MAXCLASS;' MORE MOVES'
[20] RETURN:'MY GUESS: ',,DECODE R
[21]  →0
    ∇


      ∇HOWSPLITS[□]∇
    ∇ C←HOWSPLITS WI
[1]   ⍝VERSION 4: PERFORMS SPLIT IN SINGLE MATRIX OP
[2]   ⍝ HOWSPLITS SCORES EACH WORD IN JSIEVED AGAINST WI,
[3]   ⍝ AND RETURNS A VECTOR WITH 6 ELEMENTS SHOWING SPLIT
[4]   C←+/(¯1+⍳6)∘.=+/WORDS[JSIEVED;]∈WI
    ∇
```

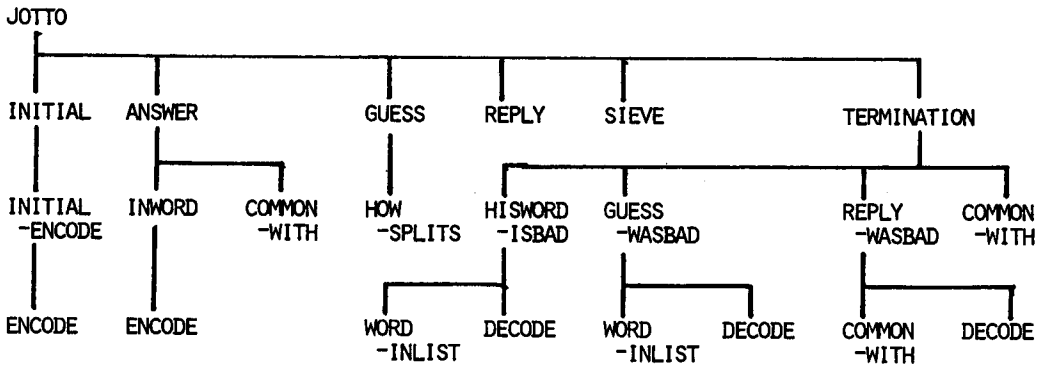
```
      ∇REPLYWASBAD[□]∇
    ∇ R←REPLYWASBAD;TURN;N
[1]   ⍝VERSION 2: DECODE 'FAST-SCAN' CODES
[2]   R←TURN←0
[3]  NEXT:→0 IF(MOVE⌊20)<TURN←TURN+1
[4]   N←HISWORD COMMONWITH MYGUESSES[TURN;]
[5]   →NEXT IF HISREPLIES[TURN]=N
[6]   'WHEN I GUESSED: ';DECODE MYGUESSES[TURN;]
[7]   'YOU SAID ';HISREPLIES[TURN];' LETTERS IN COMMON'
[8]   'THERE ARE ACTUALLY ';N;' LETTERS IN COMMON WITH: ';DECODE
      HISWORD
[9]   'THEREFORE, I CLAIM VICTORY'
[10]  R←1
    ∇

      ∇WORDINLIST[□]∇
    ∇ R←WORDINLIST W
[1]   R←∨/WORDS∧.=W
[2]   ⍝  RETURNS 1 IF W IS IN WORDS
    ∇

      ∇GUESSWASBAD[□]∇
    ∇ R←GUESSWASBAD;TURN
[1]   ⍝VERSION 2: DECODE 'FAST-SCAN' CODES
[2]   ⍝ SUBMODULES: WORDINLIST, DECODE
[3]   ⍝ GLOBAL VARS: MOVE,HISGUESSES
[4]   R←TURN←0
[5]  NEXT:→0 IF(20⌊MOVE)<TURN←TURN+1
[6]   →NEXT IF WORDINLIST HISGUESSES[TURN;]
[7]   R←1
[8]   'ON TURN ';TURN;' YOU SAID: ';DECODE HISGUESSES[TURN;]
[9]   '  WHICH IS NOT IN THE WORDS LIST'
[10]  'THEREFORE, I CLAIM VICTORY'
    ∇

      ∇HISWORDISBAD[□]∇
    ∇ R←HISWORDISBAD
[1]   ⍝VERSION 2: HAS TO DECODE 'FAST-SCAN' CODES
[2]   ⍝ SUBMODULES: WORDINLIST,DECODE
[3]   R←0
[4]   'MY HIDDEN WORD WAS: ';DECODE MYWORD
[5]   HISWORD←INWORD 'WHAT IS YOUR WORD?'
[6]   →0 IF WORDINLIST HISWORD
[7]   R←1
[8]   'YOU CHEATED.  ';DECODE HISWORD;' IS NOT LEGAL'
[9]   'THEREFORE, I CLAIM VICTORY.'
    ∇

      ∇TERMINATION[□]∇
    ∇ TERMINATION;HEWINS;IWIN
[1]   'I BELIEVE THE GAME IS OVER'
[2]   'BUT I HAVE TO CHECK THE PLAYS'
[3]   →OVER IF HISWORDISBAD
[4]   →OVER IF GUESSWASBAD
[5]   →OVER IF REPLYWASBAD
[6]   →(MOVE>20)/DRAW
[7]   HEWINS←((MYWORD COMMONWITH HISGUESSES[MOVE;])=5)
[8]   IWIN←(HISREPLIES[MOVE]=5)
[9]   →DRAW IF IWIN∧HEWINS
[10]  →LOSS IF HEWINS
[11]  ⍝
[12] WIN:'I CLAIM VICTORY'
[13]  →OVER
[14]  ⍝
[15] LOSS:'IF YOU CLAIM IT, YOU WIN'
[16]  →OVER
[17]  ⍝
[18] DRAW:'IT SEEMS TO BE A DRAW'
[19] OVER:'TO PLAY AGAIN TYPE:'
[20]  '     JOTTO'
    ∇
```


JOTTO
INITIAL
ANSWER
GUESS
REPLY
SIEVE
TERMINATION
INITIAL -ENCODE
INWORD
COMMON -WITH
HOW -SPLITS
HISWORD -ISBAD
GUESS -WASBAD
REPLY -WASBAD
COMMON -WITH
ENCODE
ENCODE
WORD -INLIST
DECODE
WORD -INLIST
DECODE
COMMON -WITH
DECODE