Computer Language Dec. 1986

THEORY & PRACTICE

Scissors, paper, stone: A tournament of Schemes

By Brian Hayes

In an old episode of the British television series *Doctor Who*, the Doctor and his companion Romana are off in a corner of the galaxy trying to keep peace between two races of robots.

The Doctor observes that war among perfectly rational and deterministic beings can never end—or even properly get started—because each side will always avoid fighting unless it is sure of winning. When a battle is imminent, each army will assess the other's strength, and the weaker will withdraw before a shot is fired.

To illustrate his point, the Doctor suggests a few rounds of scissors-paper-stone. In this children's game, as most readers will remember, two players stand face to face, each concealing one hand behind his or her back. At a signal the players bring their hands forward, revealing either two fingers (scissors), an open hand (paper), or a closed fist (stone). The winner is determined by the following circular relations: scissors cut paper, paper wraps stone, stone dulls scissors.

When the Doctor and Romana play, they each win a few throws, and the lead is traded back and forth. When two robots try the game, the result is a scoreless tie: on each throw both robots make the same choice, playing scissors against scissors, paper against paper, and stone against stone. Finally the Doctor takes on one of the robots and wins consistently, proving yet again that logic must bow to intuition.

The notion that waging war requires a measure of irrationality seems plausible enough, but the robots' sorry performance in scissors-paper-stone leaves me unconvinced. I suspect even a very dull automaton could avoid the stalemate of perpetual ties. And with only rudimentary analytic skills a robot should be able to hold its own against a human player.

In any case, there is no need to take the Doctor's word for it, or mine. One can simply build a program to play the game, and put it to the test directly.

The project provides an opportunity to explore PC Scheme, Texas Instruments' new implementation of the Scheme programming language. Scheme is a dialect of LISP, and in my view a particularly elegant one. A program for playing scissors-paper-stone illustrates some of Scheme's most interesting features.

Playing at random

Scissors-paper-stone is an unusual game: it has a perfect defensive strategy but no reliable offense. In other words, you can avoid losing but you cannot be sure of winning. The unbeatable defense consists in choosing your moves completely at random, so that your opponent cannot predict what you will do next. On any given throw you are equally likely to play scissors, paper, or stone; in the long run, you can expect to win a third of all the throws, lose a third, and tie a third. No other strategy can be guaranteed to do better against all possible opponents.

The Doctor might well attribute his easy victory over the robots to their inability to play randomly. By definition, a deterministic machine cannot do anything at random. All the actions of a robot or a computer are specified by an algorithm, and in principle they can be predicted in full detail. The crucial phrase, however, is "in principle"; unless you happen to know the machine's algorithm and its initial state, outguessing a computer is exceedingly difficult. Perhaps a Time Lord from the planet Galifrei can do it, but few earthlings can.

Although a computer cannot act randomly, it can readily produce a pseudorandom sequence of moves—one that has all the statistical properties of a truly random sequence and hence appears to be patternless. As a matter of fact, computers are a good deal better than people at simulating randomness. Without resorting to external aids such as dice to roll or coins to flip, a human player has a hard time excluding all traces of pattern from a series of moves. People tend to make the sequence "too random," avoiding all repetition of the same move. A computer has no such unconscious biases (unless the programmer implants them).

A computer generating pseudorandom moves for scissors-paper-stone should be able to achieve a draw with a human player, but can a machine go beyond this level of play and attempt to win a match? This is a question best answered by experiment, but before considering it in detail some further analysis of the game will be helpful.

An interesting property of the random-play strategy is that the opponent's method of choosing moves has no bearing at all on the outcome of a match. Even a strategy that seems quite foolish, such as always making the same move, works as well as any other; on the average each player will win a third of the throws and the rest will be ties.

Random play, however, is the only strategy that has this property. Once a player abandons random choice, the situation grows more complicated.

Suppose you are making strictly random moves when you notice that your opponent is playing paper slightly more often than either scissors or stone. You could ignore this bias and still be confident of a tie, or you could attempt to exploit your knowledge. By giving scissors a slightly higher weight in your own choice of moves, you would skew the probabilities in your favor and might hope to win.

But making a bid for victory is a risky business. If your opponent notices the change in the statistics of your moves, he or she can begin to play stone more often and thereby turn your strategy against you.

Whereas random choice makes all other strategies irrelevant, any deviation from randomness turns the game into a contest of pattern recognition. A player aiming to win must examine the history of the game, hoping to find some pattern that offers a clue to what the opponent will do next. If you can predict the next move with certainty, you can win every throw. In general, certainty is out of reach, but detecting even a slight bias in the probabilities can be helpful. For example, if all you know is that paper is a little less likely to be played next than either scissors or stone, you can confidently choose stone as your own move; the odds are it will win or tie.

The more interesting programs for playing scissors-paper-stone make judgments on the basis of a pattern analysis. They accumulate information about their opponent's habits and tendencies, then put this knowledge to work in choosing their own moves. It turns out that some surprisingly simple programs perform quite well.

The scheme of things

A program to play scissors-paper-stone is not the kind of software that begins with a full specification and develops through top-down design. The point of writing the program is not simply to get answers but also to find out what questions are worth asking. Such a program must evolve through experiment and exploration.

Scheme is a language well suited to the exploratory style of programming. This is not to say that you can begin without forethought, charge ahead blindly, and never have to revise a line. With care, however, a Scheme program can be built out of small and versatile elements that fit together in many ways.

Scheme was devised in the 1970s by Guy Lewis Steele Jr. and Gerald Jay Sussman of the Massachusetts Institute of Technology, Cambridge, Mass. At first glance a Scheme program looks much like code in any other LISP dialect: both data structures and program statements are represented as lists, which are often nested to form lists of lists. As in other varieties of LISP, there are lots of parentheses.

Under the skin, however, Scheme differs fundamentally from other LISPs. Two unusual features of the language deserve mention here: the use of block structure, with lexical scope, and the idea of first-class procedures.

Block structure and lexical scope should hold no mysteries for anyone familiar with Pascal, Ada, or one of the other offspring of ALGOL-60. In these languages the names of variables and procedures can be made local to a block of code and thus invisible outside that block. The rules that determine the scope of a name are said to be lexical because the name's meaning can be deduced from where it appears in the program text. In most dialects of LISP (Scheme and Common LISP are the major exceptions) names have dynamic scope, and their meaning can be determined only at run time.

In Pascal the scope of a local variable is an entire procedure. Scheme allows names to be confined to an even smaller compass. A common way of introducing local variables is the *let* statement, as in:

```
(let ((var expr)) stmts . . . )
```

Here *var* is assigned the initial value returned by *expr* and is accessible to any statements within the parentheses that delimit *let*. Outside of those parentheses, however, *var* does not exist.

The notion of first-class procedures is perhaps the sweetest innovation in Scheme, but to see its significance one must first recognize that the procedures of other languages are in fact second-class citizens.

In standard Pascal, for example, a procedure can be passed as an argument to another procedure, but it cannot be returned as the value of a function or stored as the value of a variable; there can be no arrays of procedures. Even in LISP, procedures require special treatment under some circumstances.

Scheme abolishes all restrictions on the handling of procedures. Indeed, *every* object in Scheme has first-class status: anything that can be done with a simple value such as a number can also be done with a procedure, the environment in which a procedure executes, or even the default "future" of a computation.

First-class procedures are not merely theoretical niceties or tricks useful only in cute, self-modifying programs. They add much to the expressive power of Scheme, and they promote a distinctive style of program development. Procedures whose returned values are other procedures are at the heart of the scissors-paper-stone program.

The referee

In an exploratory program, versatility is at a premium. What is needed in the scissors-paper-stone program is a convenient means for creating a variety of robot players and staging games between a robot and a human player or between two robots.

One obvious decision is to make each player an independent procedure. When one of the player procedures is called, it is expected to return a legal move: either scissors, paper, or stone. The algorithm the player uses to choose the move is an internal matter and is hidden from the rest of the system. The human player can be represented by a procedure much like any other, except that when it is polled for a move it gets it by reading the keyboard.

Along with the players, a master procedure is also needed to act as a kind of referee. It must call the two player procedures, collect the responses, determine the winner of each throw, keep score, and report the results. Because the referee has a central role in the program, it is a logical place to start in laying out the structure of the code.

In a quick first draft, the referee might be a procedure that accepts two arguments, namely the two player procedures competing in a game. Each time the referee is called, it calls the two players in turn, compares the moves to decide the winner, and updates the total score. The value returned by the referee is a list of five items: the two moves, the winner of the throw, and the two players' current scores.

This plan is on the right track, but it has a few subtle problems. Consider the matter of keeping score. If the variables that hold the two scores were local to the referee procedure, they would be reset to zero at the start of each throw. The scores could be made global variables, but that also has certain drawbacks. In the first place, it invites cheating: a player can easily win every game if it has free access to the scoreboard. More seriously, with global score variables ending one game and starting a new one becomes awkward. We then *want* the scores reset to zero, and some separate signal or procedure would be needed to do it.

Scheme offers a clever solution. Instead of building the referee procedure directly, we can write a procedure that creates a new, independent referee for each game.

I have named the referee-building procedure *make-game*. Each time it is called, it creates and initializes a few local variables—including variables for the two scores—and then returns as its value a procedure of no arguments. The latter procedure is the actual referee. Because it is within the lexical scope of *make-game*, the referee can access the score variables, but those variables retain their values between calls to the referee.

Typically, *make-game* would be invoked by a statement like the following:

```
(define game
   (make-game player1 player2)).
```

Here a new variable *game* is declared and assigned the value that results from evaluating *(make-game player1 player2)*. As already noted, the value of this expression is a procedure of no arguments whose lexical environment includes the two score variables. Evaluating the statement *(game)* now invokes the synthesized procedure, which calls *player1* and *player2*, decides the winner, updates the scores, and posts the results. Note that *make-game* is called once for each game, whereas *game* is called once for each throw.

The players

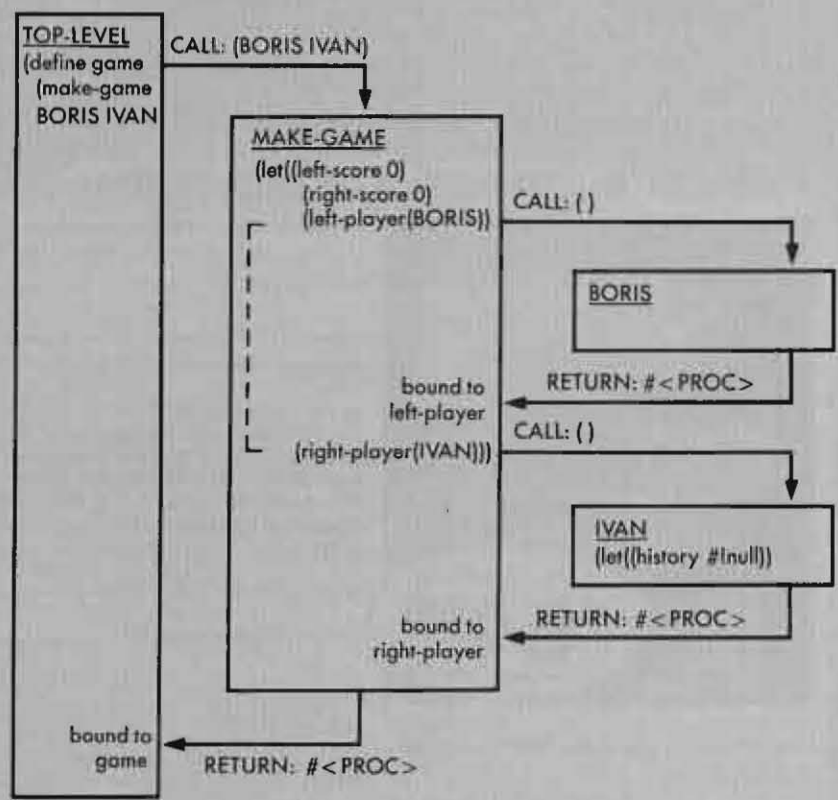
Problems similar to those encountered in *make-game* arise in building the player procedures.

Suppose a player named Ivan chooses moves by consulting a history of all the moves its opponent has made so far in a game. The history cannot be stored in a local variable, because the record would be started fresh on each throw. Global storage is also unacceptable. Again it would make the player vulnerable to a cheating opponent (which could alter Ivan's memory!) and would require special measures for initialization. In addition, the procedure might interfere with its own operation. Consider what would happen if Ivan were playing two games simultaneously, or if Ivan were pitted against Ivan in a single match.

The answer is again to write a procedure that creates a procedure. The top-level procedure, which is the one named Ivan, is called once at the beginning of a game. It sets up the necessary variables and then returns another procedure, which does the actual playing. When the latter procedure is called (once for each throw), it selects and returns a move.

The relations between the referee and player procedures are diagramed in Figures 1 and 2. Organizing the program in this way makes each game and each player a completely isolated entity. The modules can communicate only through arguments passed to a procedure and values returned

First-class procedures



The program for playing scissors-paper-stone relies on several procedures whose returned values are themselves procedures. The user invokes the procedure *make-game*, naming the players Boris and Ivan as arguments. *Make-game* initializes the local variables *left-score* and *right-score*, then calls Boris and Ivan. The values returned by the two players are procedures, which are bound to the variables *left-player* and *right-player*. Finally *make-game* returns another procedure, which becomes the value of the variable *game*.

Figure 1.

by it. When Ivan takes part in a game, it is not the globally defined Ivan that plays but rather an instance of the Ivan algorithm, which lives in an environment separate from any other instances of Ivan that might be present in the system. The sequence of statements:

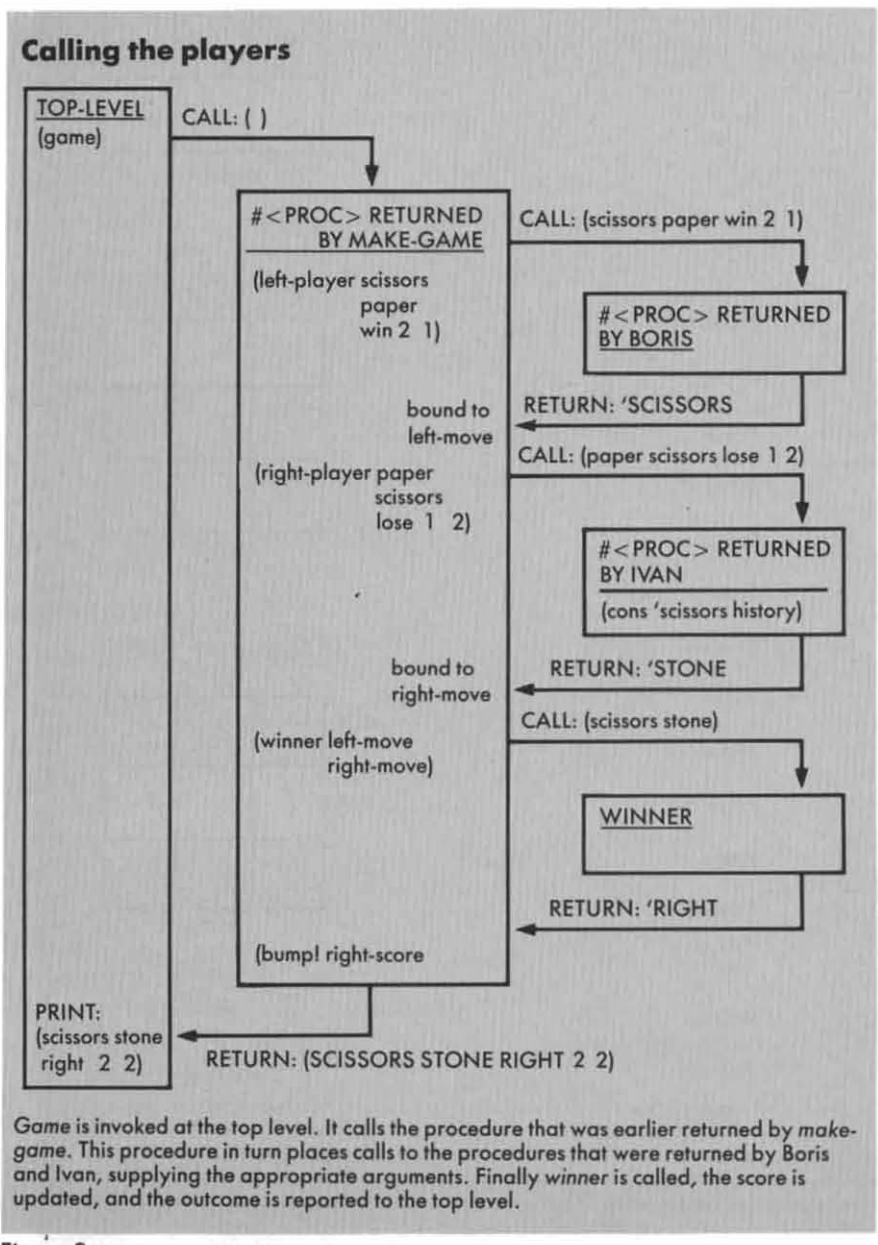
```
(define g1 (make-game Ivan Ivan))
(define g2 (make-game Ivan Fred))
```

would set up two games, named *g1* and *g2*, played by three instances of Ivan and one instance of a player called Fred. Even if *g1* and *g2* are executed alternately, there can be no interference between the games or between the players.

One other issue in the construction of the player procedures requires comment. If a player is to keep track of a game's history, it must somehow be given access to that history. There are several ways this might be accomplished. The possibility of storing the sequence of moves in a global variable can be dismissed immediately, for reasons that should already be apparent. Another approach would be to make two calls on each player for each throw: the first call would request a move and the second would report the outcome of the throw. Or a player might consist of two linked procedures, one to generate a move and the other to accept a report.

The method I have adopted is to make each player a single procedure that receives a single call for each throw. Arguments passed during the call supply information about the results of the previous throw. The one disadvantage of this arrangement is that special provisions must be made for the first throw in a game, but the burden of extra code is not great.

Five arguments accompany each call to a player. They tell the player its own previous move; its opponent's previous



Game is invoked at the top level. It calls the procedure that was earlier returned by *make-game*. This procedure in turn places calls to the procedures that were returned by Boris and Ivan, supplying the appropriate arguments. Finally *winner* is called, the score is updated, and the outcome is reported to the top level.

Figure 2.

move; whether it won, lost, or tied; its own score; and its opponent's score. Naturally, the player does not have to make use of all this information; indeed, some players ignore all of it.

Strategies

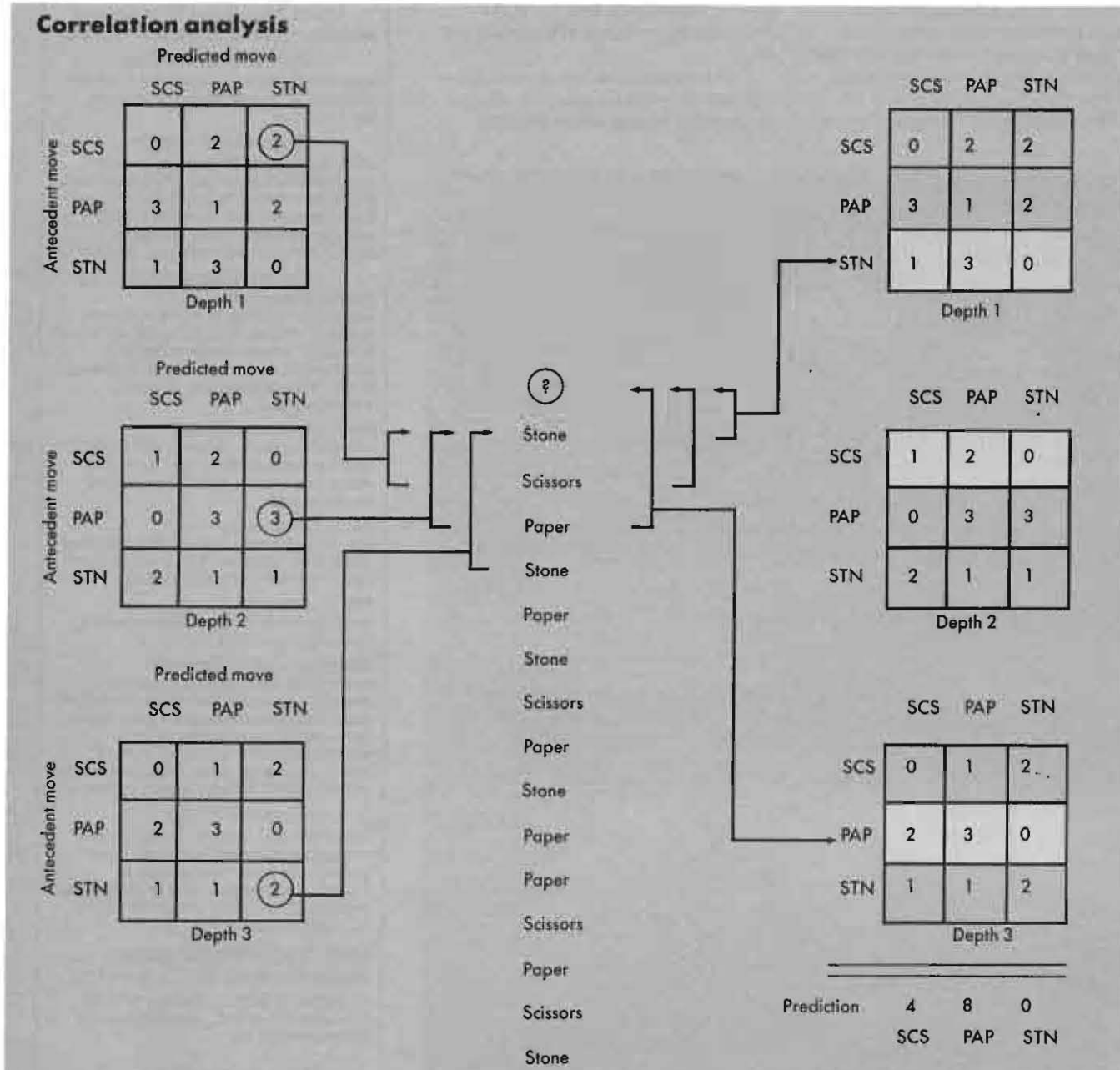
Of the dozen players I have developed and tested, the two simplest are named Adam and Boris.

Adam is the proxy procedure for the human player; when it is asked to select a move, it simply awaits instructions from the keyboard.

Boris chooses its moves randomly. To do so it makes use of the built-in Scheme procedure *(random n)*, which generates a pseudorandom integer between 0 and $n - 1$. Thus *(random 3)* yields one of the three integers 0, 1, and 2, and these values can be associated with the legal moves scissors, paper, and stone.

From our analysis of random play, one would expect Boris to have indifferent success against all comers. Experiment seems to confirm this prediction. In a series of 55 games against 11 opponents, Boris won 25 games and lost 30; in another series of 60 games against four opponents, it won 29 and lost 31. Although I have not attempted a statistical analysis, the record of wins and losses has the look of a random distribution. On occasion Boris defeated some of the strongest players and lost to some of the weakest.

In my own games against Boris, I sometimes relieved the tedium of thinking up new moves by repeating Boris's own previous move. It was a simple matter to create a robot player that turned this lazy habit into an algorithm; I called the player Claude. It turns out the copycat strategy works well enough against Boris— but then *any* strategy has an equal chance against Boris. Against most other players Claude failed miserably; in the 55-game tournament it scored 17 victories.



Kurt, the strongest of the robot players, attempts to predict its opponent's actions by detecting correlations between a given move and the subsequent moves. The correlations are stored in an array of tables. In the state of the system shown here the four most recent moves are stone, paper, scissors, and stone. From this sequence the following correlations can be deduced: scissors predicts stone will be the next move, paper predicts stone will be played two moves later, and stone predicts stone will be repeated after three moves. The correlations are recorded by incrementing the appropriate table entries (*left*). Then Kurt attempts to predict the next move by consulting the appropriate table for each of the past three moves.

Figure 3.

The crudest sort of pattern recognition is frequency analysis. A player keeps track of the number of times its opponent has played scissors, paper, and stone, assumes the same frequency distribution will be maintained in the future, and chooses its move accordingly.

For example, if the opponent has favored scissors, then stone is the recommended move. The player David relies on this strategy. In the tournament it generally either shut out its opponent 5-0 or lost 0-5; overall it had 31 victories.

The strangest pair of players is Edgar and Fred. The rationale for Edgar's algorithm is the observation that when people try to play randomly, they tend to avoid making the same move twice in a row. They create a sequence of moves that has fewer and shorter runs than a truly random sequence would. Edgar exploits this tendency in choosing its own moves. If the opponent has just played scissors, Edgar assumes that scissors is the least likely next move.

Out of curiosity, I also wrote a converse procedure, Fred, whose working assumptions are the opposite of Edgar's. If scissors has just been played, Fred bets it will appear again. I could see no argument in favor of this strategy, and I fully expected Fred to be the whipping boy of all the players. The results came as a surprise. Both Edgar and Fred scored three wins against Boris, which of course is a matter of chance. Edgar also defeated Claude 5-0, but lost all 45 games against the other players. Fred, in contrast, finished a strong fourth in the tournament, with 38 wins overall.

George is a trick player. It makes an initial run, repeatedly playing the same move, in an attempt to establish a lead; if it succeeds, it switches to random play. The rationale for this strategy is similar to the one for Edgar: just as people tend to avoid long runs in their own play, they tend to doubt that an opponent's run will be continued. The results were similar to Edgar's: George won only 14 games.

Herman's strategy is to reward success. Each time a move wins a throw, that move is given a higher weight in the choice of subsequent moves. Curiously, Herman's tournament record is closely correlated with that of David, the player whose moves are based on frequency analysis. For the most part they won and lost the same matches and by the same 5-0 or 0-5 margin.

Ivan has already been mentioned as a procedure that maintains a historical record of each game. It uses the record to count how many times in the past 10 throws the opponent played each of the three legal moves; it then chooses a move on the assumption that the opponent will tend to equalize the distribution. For example, if scissors has been played three times, paper twice, and stone five times, Ivan will expect to see paper or scissors

next. This is another variation of the Edgar strategy, and like Edgar it performed dismally: Ivan won seven games out of 55.

Pattern analysis

The players introduced so far all rely on simple rules of thumb to pick a move. At best they are clever rather than smart. Players that attempt a deeper analysis of the game should be able to do better.

The procedure Jim chooses a move on the basis of a first-order correlation analysis. On each throw the opponent's move is appended to a list of past moves, which thus grows continuously throughout the game. This archive is then consulted to predict the opponent's next move.

Suppose the most recent move is paper. Jim looks through the archive in chronological order, taking note of each time paper was played and of what move followed paper. The probabilities for the next move are assumed to match the historical distribution. In other words, if the opponent has shown a tendency to play stone following paper, then stone is the move to expect.

The idea behind correlation analysis is to detect the patterns that infect human players' moves no matter how hard they try to suppress them. The algorithm seems to work well against both machines and people. Jim defeated all but two of the other robot players and had a total of 44 wins.

A first-order correlation analysis looks for a connection between consecutive moves. There could also be a correlation between a given move and the move made two, three, or more throws later. In other words, the fact that move 0 is scissors might be a strong indication that move 2 will be stone or move 3 will be scissors again. The player Kurt searches for such correlations to a depth of eight throws.

Kurt could employ the same method for detecting correlations as Jim, but Kurt would have to make eight passes through the entire history of moves on every throw and would therefore take at least eight times as long. A more efficient approach is to record the correlations in a table as each new move is reported. For each depth, or correlation distance, the table has nine entries, which correspond to the nine possible combinations of antecedent and predicted moves. The way the tables are compiled and used for prediction is illustrated in Figure 3.

Kurt proved to be a highly successful—indeed, formidable—player. Apart from a chance defeat by Boris, it lost against only one other player, namely Jim. Possible reasons for this one loss will be discussed later.

Both Jim and Kurt treat a series of moves as if it were a stream of symbols generated in isolation, not in the give-and-take of a two-player game. In practice a player's moves might depend not only on his or her own past actions but also on what the opponent has done.

Thus a logical extension to Kurt's correlation analysis is to include both sides' moves in the history. Lars makes this extension. Like Kurt, it searches for correlations to a depth of eight throws, but because each throw includes two moves the depth of search is 16 moves.

Lars was a great disappointment. It is a ponderous and elaborate procedure that does a great deal of careful analytic work

Tournament results

	Boris	Claude	David	Edgar	Fred	George	Herman	Ivan	Jim	Kurt	Lars	Murray	Wins
Boris	—	1	3	2	2	2	2	3	4	4	2	0	25
Claude	4	—	1	0	3	0	1	5	1	0	2	0	17
David	2	4	—	5	0	5	5	5	0	0	5	0	31
Edgar	3	5	0	—	0	0	0	0	0	0	0	0	8
Fred	3	2	5	5	—	5	5	5	2	0	2	4	38
George	3	0	0	5	0	—	0	5	0	0	1	0	14
Herman	3	4	0	5	0	5	—	5	0	0	2	1	25
Ivan	2	0	0	5	0	0	0	—	0	0	0	0	7
Jim	1	4	5	5	3	5	5	5	—	4	5	2	44
Kurt	1	5	5	5	5	5	5	5	1	—	5	3	45
Lars	3	3	0	5	3	4	3	5	0	0	—	0	26
Murray	5	5	5	5	1	5	4	5	3	2	5	—	45
Losses	30	33	24	47	17	36	30	48	11	10	29	10	

Adam: The human player; gets moves from the keyboard.
Boris: Chooses moves at random with uniform distribution.
Claude: Reproduces opponent's most recent move.
David: Assumes opponent's moves have constant frequency distribution.
Edgar: Assumes opponent will not repeat the same move.
Fred: Converse of Edgar; assumes opponent will repeat.
George: Makes initial run of repeated moves, then plays randomly.
Herman: Chooses the move that has won most frequently in the past.
Ivan: Assumes uniform distribution over a span of 10 moves.
Jim: Does first-order correlation analysis of opponent's moves.
Kurt: Does eighth-order correlation analysis of opponent's moves.
Lars: Does 16th-order correlation analysis of both sides' moves.
Murray: A "metaplayer"; chooses moves suggested by consultant players.

Each player completed five games against the other 11 players. Reading across a row gives the number of wins scored in each match; reading down a column gives the number of losses. A game was won by the first player to reach 50 points.

Table 1.

(and spends a fair amount of time) in choosing each move. Nevertheless, it failed to improve on Kurt and also fell behind four other players. The full explanation is not clear, but part of the answer may be that the correlations Lars looks for simply do not exist. As a result the signal detected by Kurt is obscured by the noise of random coincidences.

The final player in this catalog has no strategy of its own. It is Murray the metaplayer, which draws on the collective wisdom of several other players in choosing its moves. First-class procedures are essential to Murray's operation. It works like this: when Murray is first called, it places calls in turn to all the procedures on a list of consultant players. For each throw thereafter, Murray passes on the information supplied by the referee and collects the moves suggested by the consultants. The move submitted by the consultant with the best winning percentage is returned to the referee as Murray's move.

In the 12-player tournament Murray's list of consultants included the nine players from Boris through Jim. (Kurt and Lars were kept off the list because they are too slow. Murray itself was also excluded, and the reason is worth a moment's thought: what would happen if Murray were on the list of procedures called by Murray?) With all this talent on call, the metaplayer did quite well, winning 45 games and tying with Kurt for first place.

Match point

The results of the tournament are summarized in Table 1. Who takes the grand prize? It is difficult to say. Kurt and Murray won the most games, but Kurt was beaten by Jim, and Murray was beaten by both Kurt and Fred. Kurt and Jim lost to Boris. There is a tangle of nontransitive relations here: just as scissors beats paper beats stone beats scissors, Kurt beats Murray beats Jim beats Kurt.

To settle the question I organized a playoff tournament for the four leaders—Kurt, Murray, Jim, and Fred—with Boris included as a kind of mindless ballast. I also changed the playing conditions. In the main tournament each match consisted of five games, and a game was won by the first player to reach 50 points. Games of this length are probably adequate to test most of the strategies, but Kurt and Lars, searching for long-range correlations, may have had too little chance to show their prowess. In particular, the short games may explain why Kurt and Lars lost to Jim, the player with the similar but simpler strategy of examining only first-order correlations. In the playoff a match lasted for 15 games of 150 points each.

The longer games made a difference. Kurt emerged the clear victor, winning 51 of 60 games. The scores are given in Table 2.

In roughly 150 matches among a dozen robot players the kind of stalemate envisioned by the Doctor occurred just once. George and Claude had a tournament record against each other of 0-0; neither of them was able to win more than one throw, and their games would have gone on indefinitely if the referee had not intervened. Recall that George makes the same move repeatedly unless it gets the lead, and Claude copies its opponent's last move. I should have foreseen how these strategies would interact; no doubt the Doctor would have.

So much for robot wars. Where does the human player fit into the ranking? I cannot give a firm answer, for two reasons. In the first place, I doubt my own qualifications to carry the standard for all humanity in this competition. (I tried to recruit an Intergalactic Grandmaster, but the Doctor was not in.) Second, although writing programs to play scissors-paper-stone is moderately diverting, actually playing the game is not a whole lot of fun. After the first few hundred throws it wears thin. I have not played enough games to have much confidence in my judgment.

The championship playoff

	Boris	Fred	Jim	Kurt	Murray	Wins
Boris	—	7	6	7	9	29
Fred	8	—	9	0	11	28
Jim	9	6	—	0	11	26
Kurt	8	15	15	—	13	51
Murray	6	4	4	2	—	16
Losses	31	32	34	9	44	

Boris and the four leaders of the tournament played 15 games of 150 points against each opponent. The longer games evidently favored the correlation analysis done by Kurt.

Table 2.

Nevertheless, I have at least tried my hand against all the players, and the following observations seem worth reporting. As might be expected, Boris remains imperturbable, no matter who or what it plays: it cannot be beaten, but it never wins decisively either.

A few of the players, notably Claude and George, are absurdly easy to defeat; their strategies are too transparent. For some other players there is a counter-strategy that always works, but it is not quite as easy to discover (unless you happen to be the creator of the program). Edgar and Ivan, for example, are vulnerable to a player who makes the same move on every throw, but they perform reasonably well against a person trying to play randomly. Fred and Herman can also be beaten if you know the trick.

There is not much glory, however, in defeating robot players that have already been thoroughly trounced by their own kind. Sportsmanship demanded that I take on the best of the programs. Feeling that the honor and pride of my species were at stake, I issued a challenge to Kurt: 10 games of 100 points each. Kurt bowed from the waist and accepted the challenge. We met at dawn. There is no need to dwell on the score. It is a silly game anyway, and it has nothing to do with real intelligence. ■

The listings that accompany this column include the core procedures of a scissors-paper-stone program: the referee, some of the players, and a few essential auxiliary functions. A more complete listing is available on the COMPUTER LANGUAGE *Bulletin Board Service and CompuServe forum. It includes two shell procedures that make it more convenient to play individual games and run a multiplayer tournament.*

```
;;;Auxiliary functions
(syntax (bump! var)
  (set! var (add1 var)))

(define (list-binop op L1 L2)
  (cond ((null? L1) L2)
        ((null? L2) L1)
        (else (cons (op (car L1) (car L2))
                    (list-binop op (cdr L1) (cdr L2))))))

(syntax (add-lists L1 L2)
  (list-binop + L1 L2))

(define (weighted-choose scs-wt pap-wt stn-wt)      ;if args are 3 2 1, then...
  (let ((roll (random (+ scs-wt pap-wt stn-wt))))     ;roll = {0 1 2 3 4 5}
    (cond ((< roll scs-wt) 'scissors)                 ;if roll = {0 1 2}
          ((< roll (+ scs-wt pap-wt)) 'paper)         ;if roll = {3 4}
          (else 'stone))))                            ;if roll = {5}

(define (expect scs-prob pap-prob stn-prob)
  (let ((scs-ex (max (- pap-prob stn-prob) 0))
        (pap-ex (max (- stn-prob scs-prob) 0))
        (stn-ex (max (- scs-prob pap-prob) 0)))
    (if (and (zero? scs-ex) (zero? pap-ex) (zero? stn-ex))
        (weighted-choose 1 1 1)
        (weighted-choose scs-ex pap-ex stn-ex))))
```

Listing 1.

```
;;;A selection of players
(define (Boris)
  (lambda (my-move opponent-move outcome my-score opponent-score)
    (weighted-choose 1 1 1)))

(define (Claude)
  (lambda (my-move opponent-move outcome my-score opponent-score)
    (if (not opponent-move)
        (weighted-choose 1 1 1)
        opponent-move)))

(define (Edgar)
  (let ((scs 1) (pap 1) (stn 1))
    (lambda (my-move opponent-move outcome my-score opponent-score)
      (case opponent-move
        (scissors (set! scs 1) (bump! pap) (bump! stn))
        (paper    (set! pap 1) (bump! scs) (bump! stn))
        (stone    (set! stn 1) (bump! scs) (bump! pap)))
      (expect scs pap stn))))

(define (Fred)
  (let ((scs 1) (pap 1) (stn 1))
    (lambda (my-move opponent-move outcome my-score opponent-score)
      (case opponent-move
        (scissors (bump! scs) (set! pap 1) (set! stn 1))
        (paper    (bump! pap) (set! scs 1) (set! stn 1))
```

Listing 2. *(Continued on following page)*

```
        (stone    (bump! stn) (set! scs 1) (set! pap 1)))
      (expect scs pap stn))))

(define (Herman)
  (let ((scs 1)(pap 1)(stn 1))
    (lambda (my-move opponent-move outcome my-score opponent-score)
      (if (eq? outcome 'win)
          (case my-move
            (scissors (bump! scs))
            (paper    (bump! pap))
            (stone    (bump! stn))))
      (weighted-choose scs pap stn))))

(define (Kurt)
  (let* ((history #!null)                                  ;let* for sequence
         (max-depth 8)
         (correlations (make-vector (add1 max-depth))))
    (define (corr-init depth)                              ;loop to init vector
      (cond ((> depth max-depth) #!null)
            (else (vector-set! correlations
                               depth (make-table '(scissors paper stone)
                                                 '(scissors paper stone) 0))
                  (corr-init (add1 depth)))))
    (define (corr-ref ante post depth)
      (table-ref (vector-ref correlations depth) ante post))
    (define (corr-bump! ante post depth)
      (table-bump! (vector-ref correlations depth) ante post))
    (define (correlate depth)                              ;update for latest move
      (let ((predictor (list-ref history depth)))          ;move `depth' throws back
        (cond ((or (not predictor)                         ;if before the beginning,
                   (>? depth max-depth)) #!null)           ; or too deep, do nothing
              (else (corr-bump!                            ;predictor predicts car
                     predictor (car history) depth)        ; `depth' moves later
```

Listing 2. *(Continued on following page)*

```
                    (correlate (addl depth))))))      ;do next deeper layer
    (define (get-predictions depth)                    ;(subl depth) jogs
      (let ((predictor (list-ref history (subl depth)))) ; table for next move
        (cond ((or (not predictor)                     ;if no entry, use
                   (>? depth max-depth)) '(0 0 0))     ; additive identity
              (else (add-lists                                  ;sum correla-
                      (list (corr-ref predictor 'scissors depth)  ; tions for
                            (corr-ref predictor 'paper depth)     ; three move
                            (corr-ref predictor 'stone depth))    ; choices
                      (get-predictions (addl depth)))))))      ;add next layer
    (corr-init 0)                                  ;body of `Kurt' begins here
    (lambda (my-move opponent-move outcome my-score opponent-score)
      (if opponent-move (set! history (cons opponent-move history)))
      (correlate 1)
      (expect-list (get-predictions 1)))))
```

Listing 2. *(Continued from preceding page)*

```
;;;  Make-game  serves as the referee in scissors-paper-stone
;;; match. It calls each of the two contestants in turn, passing
;;; them the results of the previous throw and keeping track of
;;; the current score.

(define (make-game player1 player2)   ;players are procs, not names of procs
  (let ((left-player (player1))       ;each player called here to initialize,
        (right-player (player2))      ; returning the proc called on each throw
        (left-move #!false)
        (right-move #!false)
        (winning-side #!false)
        (left-score 0)
        (right-score 0))
    (lambda ()
      (let ((left-temp (left-player left-move            ;call with results
                                    right-move           ; of previous move
                                    (case winning-side   ; and get new move
                                      (left 'win)        ; as returned value.
                                      (right 'lose)      ; temp  variables
                                      (tie 'tie)         ; are needed to
                                      (else #!false))    ; ensure that 2nd
                                    left-score           ; player can have
                                    right-score))        ; no knowledge of
            (right-temp (right-player right-move         ; what 1st player
                                      left-move          ; is doing
                                      (case winning-side
                                        (right 'win)
                                        (left 'lose)
                                        (tie 'tie)
                                        (else #!false))
                                      right-score
                                      left-score)))
        (set! left-move left-temp)
        (set! right-move right-temp)
        (set! winning-side (winner left-move right-move))
        (case winning-side
          (left (bump! left-score))
          (right (bump! right-score)))
        (list left-move right-move winning-side left-score right-score)))))

(define (winner left-move right-move)
  (let ((win-table (vector (vector 'tie    'left   'right  'left)
                           (vector 'right  'tie    'left   'left)
                           (vector 'left   'right  'tie    'left)
                           (vector 'right  'right  'right  'tie ))))
    (define (index move)
      (case move
        (scissors 0)
        (paper    1)
        (stone    2)
        (else     3)))
    (vector-ref (vector-ref win-table (index left-move)) (index right-move))))
```

Listing 3.