CS 105—Fundamentals of Computer Programming

Fall 1998 Project

Prisoner's Dilemma

adapted from an assignment in CS 6.001 at MIT

The Project Assignments

In the final project for Com Sci 105, you will define a number of small *Scheme* procedures, similar in scope to those that you write for homework assignments. But, you will also use the procedures that you write, along with some code provided by your instructors, to perform interesting computational experiments that provide insight into the way that cooperative behavior may arise out of selfishness.

The project contains 8 problems, each of which you must solve by defining one or more procedures in *Scheme*. Hand in problems 1–4 online before midnight on Thursday, 3 December, and bring printed output to class on Friday, 4 December. Hand in problems 5–8 online before midnight on Wednesday, 9 December (the last day of classes). If you finish early, you may bring printed output to the last class. Otherwise, you may hand in printed output in the main Computer Science office, Ryerson 152, before noon on Thursday 10 December. Please print 5–6 and 7–8 on separate pages, so they can be separated and graded simultaneously by different TAs.

Problems 1, 6, and 7 require you to define *Scheme* procedures implementing strategies for the repeated Prisoner's Dilemma game, described below. Demonstrate your strategies on 2–4 tiny inputs, to show that they behave as required. Then, play your strategies against themselves and a few other interesting strategies. In problem 1, use the 2-player `play-loop` procedure provided on *Johnny Three*. In problems 6 and 7, use the 3-player procedure `play3-loop` that you define in problem 5.

Problems 2–4 and 8 require you to define *Scheme* procedures that take old strategies and numerical parameters as inputs, and create new strategies. In all cases, you need to use `lambda` expressions to describe the outputs of these strategy-transforming procedures, since the outputs are also procedures. Testing a strategy transformer is a bit more complicated than testing a strategy. You must choose a small number of different inputs to your strategy transformer, then test each of the resulting strategies. Then, play

2 or 3 of the strategies produced by your strategy transformer against other strategies.

Problem 5 requires you to modify the 2-player game supervision code provided by the instructors so that it works for 3 players. Hand in the *Scheme* code for each of the procedures that you modify. The game supervision code will be tested in problems 6–8. Be very careful, though. An error in problem 5 will ruin your work on problems 6–8.

For 10 points extra credit, you may submit a strategy to participate in a tournament described in the last section. hand in your entry online on *Johnny Three*. Do not submit printed copy. You will get credit for participation no matter how well or poorly your strategy fares, but your procedure must include the required comments and it must not be disqualified by generating an error. The deadline for tournament entries is midnight on Monday, 7 December.

The Prisoner's Dilemma: A Fable

In the mid-1920's, the Nebraska State Police achieved what may still be their finest moment. After a 400-mile car chase over dirt roads and corn fields, they finally caught up with the notorious bank robbers Bonnie and Clyde. The two criminals were brought back to the police station in Omaha for further interrogation.

Bonnie and Clyde were questioned in separate rooms, and each was offered the same deal by the police. The deal went as follows (since both are the same, we need only describe the version presented to Bonnie):

"Bonnie, here's the offer that we are making to both you and Clyde. If you both hold out on us, and don't confess to bank robbery, then we admit that we don't have enough proof to convict you. However, we *will* be be able to jail you both for one year, for reckless driving and endangerment of corn. If you turn state's witness and help us convict Clyde (assuming he doesn't confess), then you will go free, and Clyde will get twenty years in prison. On the other hand, if you don't confess and Clyde does, then *he* will go free and *you* will get twenty years."

"What happens if both Clyde and I confess?" asked Bonnie.

"Then you both get ten years," said the interrogator.

Bonnie, who had been a math major at Cal Tech before turning to crime, reasoned this way: "Suppose Clyde intends to confess. Then if I don't confess, I'll get twenty years, but if I do confess, I'll only get ten years. On the other hand, suppose Clyde intends to hold out on the cops. Then if I don't confess, I'll go to jail for a year, but if I do confess, I'll go free. So no matter what Clyde intends to do, I am better off confessing than holding out. So I'd better confess."

Naturally, Clyde employed the very same reasoning. Both criminals confessed, and both went to jail for ten years.[1] The police, of course, were triumphant, since the criminals would have been free in a year had both remained silent.

The Prisoner's Dilemma

The Bonnie and Clyde story is an example of a situation known in mathematical game theory as the "prisoner's dilemma." A prisoner's dilemma always involves two "game players," and each has a choice between "cooperating" and "defecting." If the two players cooperate, they each do moderately well; if they both defect, they each do moderately poorly. If one player cooperates and the other defects, then the defector does extremely well and the cooperator does extremely poorly. (In the case of the Bonnie and Clyde story, "cooperating" means cooperating with one's partner—i.e., holding out on the police—and "defecting" means confessing to bank robbery.) Before formalizing the prisoner's dilemma situation, we need to introduce some basic game theory notation.

A Crash Course in Game Theory

In game theory, a *two-person binary-choice game* is represented as a two-by-two matrix. Figure 1 shows a hypothetical game matrix. The two players in this case are called **A** and **B**, and the choices are called "cooperate" and "defect."

	B cooperates	B defects
A cooperates	A gets 5 B gets 5	A gets 2 B gets 3
A defects	A gets 3 B gets 2	A gets 1 B gets 1

Figure 1: A hypothetical game matrix

[1] Actually, they didn't go to jail. When they were in court, and heard that they had both turned state's witness, they strangled each other. But that's another story.

Players **A** and **B** can play a single game by separately (and secretly) choosing to either cooperate or defect. Once each player has made a choice, he announces it to the other player; and the two then look up their respective scores in the game matrix. Each entry in the matrix is a pair of numbers indicating a score for each player, depending on their choices. Thus, in Figure 1, if Player **A** chooses to cooperate while Player **B** defects, then **A** gets 2 points and **B** gets 3 points. If both players defect, they each get 1 point. Note, by the way, that the game matrix is a matter of public knowledge; for instance, Player **A** knows before the game even starts that if he and **B** both choose to defect, they will each get 1 point.

In an *iterated game*, the two players play repeatedly: thus, after finishing one game, **A** and **B** may play another. (Admittedly, there is a little confusion in the terminology here: you can think of each individual game as a single "round" of the larger, iterated game.) There are a number of ways in which iterated games may be played; in the simplest situation, **A** and **B** play for some fixed number of rounds (say, 200), and before each round they are able to look at the record of all previous rounds. For instance, before playing the tenth round of their iterated game, both **A** and **B** are able to study the results of the previous nine rounds.

An Analysis of a Simple Game Matrix

The game depicted in Figure 1 is a particularly easy one to analyze. Let's examine the situation from Player **A**'s point of view (Player **B**'s point of view is identical):

"Suppose **B** cooperates. Then I do better by cooperating myself (I receive five points instead of three). On the other hand, suppose **B** defects. I still do better by cooperating (since I get two points instead of one). So no matter what **B** does, I am better off cooperating."

Player **B** will, of course, reason the same way, and both will choose to cooperate. In the terminology of game theory, both **A** and **B** have a *dominant* choice—i.e., a choice that gives a preferred outcome no matter what the other player chooses to do. Figure 1, by the way, does *not* represent a prisoner's dilemma situation, since when both players make their dominant choice, they also both achieve their highest personal scores. We'll see an example of a prisoner's dilemma game very shortly.

To recap: in any particular game using the matrix of Figure 1, we would expect both players to cooperate; and in an iterated game, we would expect both players to cooperate repeatedly, on every round.

The Prisoner's Dilemma Game Matrix

Now consider the game matrix shown in Figure 2.

	B cooperates	B defects
A cooperates	A gets 3 B gets 3	A gets 0 B gets 5
A defects	A gets 5 B gets 0	A gets 1 B gets 1

Figure 2: Prisoner's Dilemma Game Matrix

In this case, Players **A** and **B** both have a dominant choice—namely, defection. No matter what Player **B** does, Player **A** improves his own score by defecting, and vice versa.

However, there is something odd about this game. It seems as though the two players would benefit by choosing to cooperate. Instead of winning only one point each, they could win three points each. So the "rational" choice of mutual defection has a puzzling self-destructive flavor.

The matrix of Figure 2 is an example of a prisoner's dilemma game situation. Just to formalize the situation, let CC be the number of points won by each player when they both cooperate; let DD be the number of points won when both defect; let CD be the number of points won by the cooperating party when the other defects; and let DC be the number of points won by the defecting party when the other cooperates. Then the prisoner's dilemma situation is characterized by the following conditions:

$$DC > CC > DD > CD$$
$$CC > (DC + CD)/2$$

In the game matrix of Figure 2, we have:

$$DC = 5$$
$$CC = 3$$
$$DD = 1$$
$$CD = 0$$

so both conditions are met. In the Bonnie and Clyde story, by the way, you can verify that:

$$\begin{aligned} DC &= 0 \\ CC &= -1 \\ DD &= -10 \\ CD &= -20 \end{aligned}$$

Again, these values satisfy the prisoner's dilemma conditions.

Axelrod's Tournament

In the late 1970's, political scientist Robert Axelrod held a computer tournament designed to investigate the prisoner's dilemma situation.[2] Contestants in the tournament submitted computer programs that would compete in an iterated prisoner's dilemma game of approximately two hundred rounds, using the same matrix shown in Figure 2. Each contestant's program played five iterated games against each of the other programs submitted, and after all games had been played the scores were tallied.

The contestants in Axelrod's tournament included professors of political science, mathematics, psychology, computer science, and economics. The winning program—the program with the highest average score—was submitted by Anatol Rapoport, a professor of psychology at the University of Toronto. In this problem set, we will pursue Axelrod's investigations and make up our own Scheme programs to play the iterated prisoner's dilemma game. The second (optional) part of this problem set is itself an experiment in the spirit of Axelrod's tournament: a contest of programs that play a *three-person* prisoner's dilemma game.

Before we look at the two-player program, it is worth speculating on what possible strategies might be employed in the iterated prisoner's dilemma game. Here are some examples:

All-Defect A program using the **all-defect** strategy simply defects on every round of every game.

[2]Actually, there were two tournaments. Their rules and results are described in Axelrod's book *The Evolution of Cooperation*.

Poor-Trusting-Fool A program using the **poor-trusting-fool** strategy cooperates on every round of every game.

Random This program cooperates or defects on a random basis.

Go-by-Majority This program cooperates on the first round. On all subsequent rounds, **go-by-majority** examines the history of the other player's actions, counting the total number of defections and cooperations by the other player. If the other player's defections outnumber her cooperations, **go-by-majority** will defect; otherwise this strategy will cooperate.

Tit-for-Tat This program cooperates on the first round, and then on every subsequent round it mimics the other player's previous move. Thus, if the other player cooperates (defects) on the nth round, then **tit-for-tat** will cooperate (defect) on the $(n + 1)$th round.

All of these strategies are extremely simple. (Indeed, the first three do not even pay any attention to the other player; their responses are uninfluenced by the previous rounds of the game.) Nevertheless, simplicity is not necessarily a disadvantage. Rapoport's first-prize program employed the **tit-for-tat** strategy, and achieved the highest average score in a field of far more complicated programs.

The Two-Player Prisoner's Dilemma Program

A Scheme program for an iterated prisoner's dilemma game is shown at the end of this problem set. The procedure `play-loop` pits two players (or, to be more precise, two "strategies") against one another for approximately 100 games, then prints out the scores for each of the two players.

Player strategies are represented as procedures. Each strategy takes two inputs—its own "history" (that is, a list of all of its previous "plays") and its opponent's "history." The strategy returns either the symbol `c` (for "cooperate") or the symbol `d` (for "defect").

At the beginning of an iterated game, each history is an empty list. As the game progresses, the histories grow (via `extend-history`) into lists of `c`'s and `d`'s. Note how each strategy must have its *own* history as its first input. So in `play-loop-iter`, `strat0` has `history0` as its first input, and `strat1` has `history1` as its first input.

The values from the game matrix are stored in a list named `*game-association-list*`. This list is used to calculate the scores at the end of the iterated game.

Some sample strategies are given at the end of the program. `All-defect` and `poor-trusting-fool` are particularly simple; each returns a constant value regardless of the histories. `Random-strategy` also ignores the histories and chooses randomly between cooperation and defection. You should study `go-by-majority` and `tit-for-tat` to see that their behavior is consistent with the descriptions in the previous section.

Practice Problem 0 (no write-up necessary) Do this one just for practice and understanding of the concepts in Prisoner's Dilemma. Do not hand in anything for Practice Problem 0. Use `play-loop` to play games among the five defined strategies. Play each strategy against itself (since two different players may use the same strategy), as well as playing against the other strategies. Notice how a strategy's performance varies sharply depending on its opponent. For example, `poor-trusting-fool` does quite well against `tit-for-tat` or against another `poor-trusting-fool`, but it loses badly to `all-defect`. Pay special attention to `tit-for-tat`. Notice how it never beats its opponent—but it never loses badly.

Problem 1 Write a new strategy `tit-for-two-tats`. The strategy should always cooperate unless the opponent defected on both of the previous two rounds. (Looked at another way: `tit-for-two-tats` should cooperate if the opponent cooperated on either of the previous two rounds.) Play `tit-for-two-tats` against other strategies, and against itself.

Problem 2 Write a procedure `make-tit-for-n-tats`. This procedure should take a number as input and return the appropriate `tit-for-tat`-like strategy. For example, `(make-tit-for-n-tats 2)` should return a strategy equivalent to `tit-for-two-tats`. Remember, a "strategy" is a procedure, so the value returned by `(make-tit-for-n-tats` *n*`)` is a procedure, given by a `lambda` expression.

Problem 3
a. Write a procedure `make-dual-strategy` which takes as input two strategies (say, `strat0` and `strat1`) and an integer (say, `switch-point`). `Make-dual-strategy` should return a strategy which plays `strat0` for the first `switch-point` rounds in the iterated game, then switches to `strat1` for the remaining rounds.

b. Use `make-dual-strategy` to define a procedure `make-triple-strategy` which takes as input three strategies and two switch points.

Problem 4 Write a procedure `niceify`, which takes as input a strategy (say, `strat`) and a number between 0 and 1 (call it `niceness-factor`). The `niceify` procedure should return a strategy that plays the same as `strat` except: when `strat` defects, the new strategy should have a `niceness-factor` chance of cooperating. (If `niceness-factor` is 0, the returned strategy is exactly the same as `strat`; if `niceness-factor` is 1, the returned strategy is the same as `poor-trusting-fool`.)

Use `niceify` with a low value for `niceness-factor`—say, 0.1—to create two new strategies: `slightly-nice-all-defect` and `slightly-nice-tit-for-tat`.

The Three-Player Prisoner's Dilemma

So far, all of our prisoner's dilemma examples have involved two players (and, indeed, most game-theory research on the prisoner's dilemma has focused on two-player games). But it is possible to create a prisoner's dilemma game involving three—or even more—players.

Strategies from the two-player game do not necessarily extend to a three-person game in a natural way. For example, what does `tit-for-tat` mean? Should the player defect if *either* of the opponents defected on the previous round? Or only if *both* opponents defected? And are either of these strategies nearly as effective in the three-player game as `tit-for-tat` is in the two-player game?

Before we analyze the three-player game more closely, we must introduce some notation for representing the payoffs. We use a notation similar to that used for the two-player game. For example, we let DCC represent the payoff to a defecting player if both opponents cooperate. Note that the first position represents the player under consideration. The second and third positions represent the opponents.

Another example: CCD represents the payoff to a cooperating player if one opponent cooperates and the other opponent defects. Since we assume a symmetric game matrix, CCD could be written as CDC. The choice is arbitrary.

Now we are ready to discuss the payoffs for the three-player game. We impose three rules:[3]

1) Defection should be the dominant choice for each player. In other words, it should always better for a player to defect, regardless what the opponents do. This rule gives three constraints:

$$\begin{array}{rc} DCC > CCC & \text{(both opponents cooperate)} \\ DDD > CDD & \text{(both opponents defect)} \\ DCD > CCD & \text{(one opponent cooperates, one defects)} \end{array}$$

2) A player should always be better off if more of his opponents choose to cooperate. This rule gives:

$$DCC > DCD > DDD$$
$$CCC > CCD > CDD$$

3) If one player's choice is fixed, the other two players should be left in a two-player prisoner's dilemma. This rule gives the following constraints:

$$\begin{aligned} CCD &> DDD \\ CCC &> DCD \\ CCD &> (CDD + DCD)/2 \\ CCC &> (CCD + DCC)/2 \end{aligned}$$

[3]Actually, there is no universal definition for the multi-player prisoner's dilemma. The constraints used here represent one possible version of the three-player prisoner's dilemma.

$$CDD = 0$$
$$DDD = 1$$
$$CCD = 3$$
$$DCD = 5$$
$$CCC = 7$$
$$DCC = 9$$

Problem 5 Revise the Scheme code for the two-player game to make a three-player iterated game. The program should take three strategies as input, keep track of three histories, and print out results for three players. You need to change only three procedures: `play-loop`, `print-out-results`, and `get-scores`. Give your revised procedures the new names `play3-loop`, `print-out-results3`, and `get-scores3` You also need to replace `*game-association-list*` by `*game3-association-list*` as follows:

```
(define *game3-association-list*
  '(((c c c) (7 7 7))
    ((c c d) (3 3 9))
    ((c d c) (3 9 3))
    ((d c c) (9 3 3))
    ((c d d) (0 5 5))
    ((d c d) (5 0 5))
    ((d d c) (5 5 0))
    ((d d d) (1 1 1))))
```

Problem 6 *a.* Write strategies `poor-trusting-fool-3`, `all-defect-3`, and `random-strategy-3` that will work in a three-player game. Try them out to make sure your game-supervision code is working.

b. Write two new strategies: `tough-tit-for-tat` and `soft-tit-for-tat`. `tough-tit-for-tat` should defect if *either* of the opponents defected on the previous round. `soft-tit-for-tat` should defect only if *both* opponents defected on the previous round. Play some games using these two new strategies.

Problem 7 A natural idea in creating a prisoner's dilemma strategy is to try and deduce what kind of strategies the *other* players might be using. In this problem, we will implement a simple version of this idea.

First, we need a procedure that takes three histories as arguments: call them `hist-0`, `hist-1`, and `hist-2`. The idea is that we wish to characterize the strategy of the player responsible for `hist-0`. Our procedure will return a list of three numbers: the probability that the `hist-0` player cooperates given that the other two players cooperated on the previous round, the probability that the `hist-0` player cooperates given that only one other player cooperated on the previous round, and the probability that the `hist-0` player cooperates given that both others defected on the previous round. To fill out some details in this picture, let's look at a couple of examples. We will call our procedure `get-probability-of-c`; here are a couple of sample calls.

```
> (get-probability-of-c '(c c c c) '(d d d c) '(d d c c))
(1 1 1)

> (get-probability-of-c '(c c c d c) '(d c d d c) '(d c c c c))
(0.5 1 ())
```

In the top example, the returned list indicates that the first player cooperates with probability 1 no matter what the other two players do. In the bottom example, the first player cooperates with probability 0.5 when the other two players cooperate; the first player cooperates with probability 1 when one of the other two players defects; and since we have no data regarding what happens when both other players defect, our procedure returns `()` for that case.

Write the `get-probability-of-c` procedure. Using this procedure, you should be able to write some predicate procedures that help in deciphering another player's strategy. For instance, here are two possibilities:

```
(define (is-he-a-fool? hist0 hist1 hist2)
  (equal? '(1 1 1) (get-probability-of-c hist0 hist1 hist2)))

(define (could-he-be-a-fool? hist0 hist1 hist2)
  (equal? (list true true true)
          (map (lambda (elt) (or (null? elt) (= elt 1)))
               (get-probability-of-c hist0 hist1 hist2))))
```

Use the `get-probability-of-c` procedure to write a predicate that tests whether another player is using the `soft-tit-for-tat` strategy from Problem 7. Also, write a new strategy named `dont-tolerate-fools`. This strategy should cooperate for the first ten rounds; on subsequent rounds it checks (on each round) to see whether the other players might both be playing `poor-trusting-fool`. If our strategy finds that both other players seem to be cooperating uniformly, it defects; otherwise, it cooperates.

Problem 8 Write a procedure `make-combined-strategies` which takes as input two *two-player* strategies and a "combining" procedure. `Make-combined-strategies` should return a *three-player* strategy that plays one of the two-player strategies against one of the opponents, and the other two-player strategy against the other opponent, then calls the "combining" procedure on the two two-player results. Here's an example: this call to `make-combined-strategies` returns a strategy equivalent to `tough-tit-for-tat` in Problem 7.

```
(make-combined-strategies
  tit-for-tat tit-for-tat
  (lambda (r1 r2) (if (or (equal? r1 'd) (equal? r2 'd)) 'd 'c)))
```

The resulting strategy plays `tit-for-tat` against each opponent, and then calls the combining procedure on the two results. If either of the two two-player strategies has returned `d`, then the three-player strategy will also return `d`.

Here's another example. This call to `make-combined-strategies` returns a three-player strategy that plays `tit-for-tat` against one opponent, `go-by-majority` against another, and chooses randomly between the two results:

```
(make-combined-strategies
  tit-for-tat go-by-majority
  (lambda (r1 r2) (if (= (random 2) 0) r1 r2)))
```

Extra Credit: The Three-Player Prisoner's Dilemma Tournament

As decribed earlier, Axelrod held two computer tournaments to investigate the two-player prisoner's dilemma. This year we are going to hold a three-player tournament. Past three-player tournament results have indicated that basically cooperative strategies do very well. In one tournament, for instance, the winning strategy defected only if both opponents had each defected twice in a row. And the third place strategy (out of more than fifty entries) was simply `poor-trusting-fool-3`!

You can participate this year by designing a strategy for the tournament. You might submit one of the strategies developed in the problem set, or develop a new one. The only restriction is that the strategy must work against any other legitimate entry. Any strategies that cause the tournament software to crash will be disqualified. Include a comment at the beginning of your submission containing your name, a brief mnemonic name for your strategy, and a short paragraph (2-4 sentences) explaining why you chose the strategy. You may choose a strategy in hopes that it wins, or just because its behavior is interesting in some way (e.g., you might try to sabotage one of the known strategies).

The Game Supervision Code

A printed copy of the game supervision code is attached below. This code is also in the file

`MacLab Resources:Courses:Autumn 1998:CS105:Prisoner:prisoner.scm`

on the *Johnny Three* server.

```scheme
;;; Code for Prisoner's Dilemma.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The PLAY-LOOP procedure takes as its arguments two prisoner's
;;  dilemma strategies, and plays an iterated game of approximately
;;  one hundred rounds. A strategy is a procedure that takes
;;  two arguments: a history of the player's previous plays and
;;  a history of the other player's previous plays. The procedure
;;  returns either the symbol C (for "cooperate") or D ("defect").
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (play-loop strat0 strat1)
  (define (play-loop-iter strat0 strat1 count history0 history1 limit)
    (cond ((= count limit) (print-out-results history0 history1 limit))
          (else (let ((result0 (strat0 history0 history1))
                      (result1 (strat1 history1 history0)))
                  (play-loop-iter strat0 strat1 (+ 1 count)
                                  (extend-history result0 history0)
                                  (extend-history result1 history1)
                                  limit)))))
  (play-loop-iter strat0 strat1 0 '() '() (+ 90 (random 21))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; The following procedures are used to compute and
;; print out the players' scores at the end of an
;; iterated game.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (print-out-results history0 history1 number-of-games)
  (let ((scores (get-scores history0 history1)))
    (newline)
    (display "Player 1 Score: ")
    (display (/ (car scores) number-of-games))
    (newline)
    (display "Player 2 Score: ")
    (display (/ (cadr scores) number-of-games))
    (newline)))
```

```
(define (get-scores history0 history1)
  (define (get-scores-helper history0 history1 score0 score1)
    (cond ((empty-history? history0) (list score0 score1))
          (else (let ((game (make-game (most-recent-play history0)
                                       (most-recent-play history1))))
                  (get-scores-helper (rest-of-plays history0)
                                     (rest-of-plays history1)
                                     (+ (get-player-points 0 game) score0)
                                     (+ (get-player-points 1 game) score1))))))
  (get-scores-helper history0 history1 0 0))

(define (get-player-points num game)
  (list-ref (get-point-list game) num))

(define (get-point-list game)
  (cadr (assoc game *game-association-list*)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; This list provides the "game matrix". It is used
;; by the scorekeeping procedures above.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *game-association-list*
  '(((c c) (3 3))
    ((c d) (0 5))
    ((d c) (5 0))
    ((d d) (1 1))))


;; Some selectors and constructors

(define make-game list)


(define extend-history cons)
(define empty-history? null?)

(define most-recent-play car)
(define rest-of-plays cdr)
```

```
;; A sampler of strategies

(define (all-defect my-history other-history)
  'd)

(define (poor-trusting-fool my-history other-history)
  'c)

(define (random-strategy my-history other-history)
  (if (= (random 2) 0) 'c 'd))

(define (go-by-majority my-history other-history)
  (define (count-instances-of symbol hist)
    (cond ((empty-history? hist) 0)
          ((equal? (most-recent-play hist) symbol)
           (+ 1 (count-instances-of symbol (rest-of-plays hist))))
          (else (count-instances-of symbol (rest-of-plays hist)))))
  (let ((ds (count-instances-of 'd other-history))
        (cs (count-instances-of 'c other-history)))
    (if (> ds cs) 'd 'c)))


(define (tit-for-tat my-history other-history)
  (if (empty-history? my-history)
      'c
      (most-recent-play other-history)))


;;; other possibly useful procedures

(define (get-nth-from-last-play n history)
  (list-ref history n))

(define (get-players-action player-no game)
  (list-ref game player-no))

(define (get-nth-from-last-game n my-history other-history)
  (make-game (get-nth-from-last-play n my-history)
             (get-nth-from-last-play n other-history)))
```