MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 – Structure and Interpretation of Computer Programs
Spring Semester, 2005

Project 2 – Prisoner's Dilemma

- Issued: Monday, February 21
- To Be Completed By: Friday, March 11, 6:00 pm
- Reading: Sections 2.1, 2.2.1 and 2.2.2 in *Structure and Interpretation of Computer Programs*
- Code to load for this project:
  - A link to the system code file `prisoner.scm` is provided from the Projects link on the projects section.

Purpose

Project 2 focuses on the use of higher order procedures, together with data structures. You will also further develop and demonstrate your ability to write clear, intelligible, well-documented procedures, as well as test cases for your procedures.

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 through corn fields, they finally caught up with the notorious bank robbers Bunny and Clod. The two criminals were brought back to the police station in Omaha for further interrogation. Bunny and Clod 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 Bunny):

"Bunny, here's the offer that we are making to both you and Clod. 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 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 Clod (assuming he doesn't confess), then you will go free, and Clod will get twenty years in prison. On the other hand, if you don't confess and Clod does, then *he* will go free and *you* will get twenty years."

"What happens if both Clod and I confess?" asked Bunny.

"Then you both get ten years," responded the police.

Bunny, who had been a math major at Cal Tech before turning to crime, reasoned this way: "Suppose Clod 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 Clod 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 Clod intends to do, I am better off confessing than holding out. So I'd better confess."

Naturally, Clod employed the very same reasoning. Both criminals confessed, and both went to jail for ten years (Well, 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.) 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 Bunny and Clod 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 Bunny and Clod 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, we differentiate between a *game*, and a *play*. A *game* refers to the set of possible choices and outcomes for the entire range of situations. A *play* refers to a specific set of choices by the players, with the associated outcome for that particular scenario. Thus, in game theory, a *two-person binary-choice game* is represented by a two-by-two matrix. Here is a hypothetical game matrix.

	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

The two players in this case are called **A** and **B**, and the choices are called "cooperate" and "defect." Players **A** and **B** can play a single game by separately (and secretly) choosing either to cooperate or to 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 the example above, 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; thus we refer to each iteration as a "play," which constitutes 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 by the matrix above 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. The matrix shown above, 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 re-cap: in any particular game using the above matrix, 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 following game matrix:

	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

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 second matrix 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:

$$DC > CC > DD > CD$$

$$CC > \frac{DC + CD}{2}$$

In the second game matrix, we have

$$DC = 5, \qquad CC = 3, \qquad DD = 1, \qquad CD = 0$$

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

$$DC = 0, \qquad CC = -1, \qquad DD = -10, \qquad CD = -20$$

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 (Actually, there were two tournaments. Their rules and results are described in Axelrod's book: *The Evolution of Cooperation*.). Contestants in the tournament submitted computer programs that would compete in an iterated prisoner's dilemma game of approximately two hundred rounds, using the second matrix above. 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, 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 project, we will pursue Axelrod's investigations and make up our own Scheme programs to play the iterated prisoner's dilemma game.

As part of this project, we will be running a similar tournament, but now involving a three-person prisoner's dilemma.

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:

Nasty - a program using the **Nasty** strategy simply defects on every round of every game.

Patsy - a program using the **Patsy** strategy cooperates on every round of every game.

Spastic - this program cooperates or defects on a random basis.

Egalitarian - this program cooperates on the first round. On all subsequent rounds, **Egalitarian** 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, **Egalitarian** will defect; otherwise this strategy will cooperate.

Eye-for-Eye - 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 *n*th round, then **Eye-for-Eye** will cooperate (defect) on the (*n*+1)st 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 **Eye-for-Eye** 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 provided as part of the code for this project. 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 average score of each player.

Player strategies are represented as procedures. Each strategy takes two inputs - its own “history” (that is, a list of all its previous “plays,” where for convenience we will use "c" to represent cooperate, and "d" to represent defect) and its opponent's “history.” The strategy returns either the string “c” for “cooperate” or the string “d” for “defect.” (Note that we will need to use procedures appropriate for comparing strings when we analyze these results.)

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, thus each history is stored from most recent to least recent. 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.

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

Thus, if both players cooperate, the payoff to each player is a 3, if one player cooperates and the other defects, the defecting player gets a payoff of 5, the cooperating player gets a zero payoff, if both players defect, each gets a payoff of 1.

Some sample strategies are given in the code. `Nasty` and `Patsy` are particularly simple; each returns a constant value regardless of the histories. `Spastic` also ignores the histories and chooses randomly between cooperation and defection. You should study `Egalitarian` and `Eye-for-Eye` to see that their behavior is consistent with the descriptions in the previous section.

Problem 1

To be able to test out the system, we need to complete a definition for `extract-entry`. This procedure will retrieve the payoff information from the game association list. The procedure's behavior is as follows: it takes as input a play, represented as a list of choices for each strategy (i.e., a "c" or a "d"), and the game association list. Thus a play will in this case be a list of two entries (since there are two players), each of which is the choice of action for that player. Each entry in the game association list is a list itself, with a first element representing a list of game choices, and the second element representing a list of scores (or payoffs) for each player. Thus `extract-entry` wants to search down the game association list trying to match its first argument against the first element of each entry in the game association list, one by one. When it succeeds, it returns that whole entry.

For example, we expect the following behavior:

```
(define a-play (make-play "c" "d"))

(extract-entry a-play *game-association-list*)
;Value: (("c" "d") (0 5))
```

Write the procedure `extract-entry`, and test it out using the above case `*game-association-list*`. Turn in a copy of your documented procedure and some test examples. You may want to use a diagram of the list structure to guide the creation of your code.

Problem 2

Use `play-loop` to play games among the five defined strategies. Notice how a strategy's performance varies sharply depending on its opponent. For example, `Patsy` does quite well against `Eye-for-Eye` or against another `Patsy`, but it loses badly to `Nasty`. Pay special attention to `Eye-for-Eye`. Notice how it never beats its opponent - but it never loses badly. Create a matrix in which you show the average score for tournaments pitting all possible pairings of the five different strategies: `Nasty, Patsy, Eye-for-Eye, Spastic, Egalitarian.` Describe the behavior you observe for the different strategies.

Problem 3

Games involving `Egalitarian` tend to be slower than other games. Why is that so? Use order-of-growth notation to explain your answer.

Alyssa P. Hacker, upon seeing the code for `Egalitarian`, suggested the following iterative version of the procedure:

```
(define (Egalitarian my-history other-history)
  (define (majority-loop cs ds hist)
    (cond ((empty-history? hist) (if (> ds cs) "d" "c"))
          ((string=? (most-recent-play hist) "c")
           (majority-loop (+ 1 cs) ds (rest-of-plays hist)))
          (else
           (majority-loop cs (+ 1 ds) (rest-of-plays hist)))))
  (majority-loop 0 0 other-history))
```

Compare this procedure with the original version. Do the orders of growth (in time) for the two procedures differ? Is the newer version faster?

Problem 4

Write a new strategy `eye-for-two-eyes`. The strategy should always cooperate unless the opponent defected on both of the previous two rounds. (Looked at another way: `eye-for-two-eyes` should cooperate if the opponent cooperated on either of the previous two rounds.) Play `eye-for-two-eyes` against other strategies. Describe the behavior you observe.

Problem 5

Write a procedure `make-eye-for-n-eyes`. This procedure should take a number as input and return the appropriate `Eye-for-Eye`-like strategy. For example, `(make-eye-for-n-eyes 2)` should return a strategy equivalent to `eye-for-two-eyes`. Use this procedure to create a new strategy and test it against the other strategies. Describe the observed behavior.

Problem 6

Write a procedure `make-rotating-strategy` which takes as input two strategies (say, `strat0` and `strat1`) and two integers (say `freq0 and freq1`). `Make-rotating-strategy` should return a strategy which plays `strat0` for the first `freq0` rounds in the iterated game, then switches to `strat1` for the next `freq1` rounds, and so on. (Hint: you may find it useful to think about the `remainder` procedure in order to decide which strategy to use at each iteration.) Test it against other strategies and describe the performance.

Problem 7

Write a new strategy, `make-higher-order-spastic,` which takes a list of strategies as input. It returns a new strategy that loops through this list of strategies, using the next one in the list for each play, and then starting again at the beginning of the list when it has used all the strategies. Test this new strategy against other strategies and describe the performance.

Problem 8

Write a procedure `gentle`, which takes as input a strategy (say `strat`) and a number between 0 and 1 (call it `gentleness-factor`). The `gentle` procedure should return a strategy that plays the same as `strat` except: when `strat` defects, the new strategy should have a `gentleness-factor` chance of cooperating. (If `gentleness-factor` is 0, the return strategy performs exactly the same as `strat`; if `gentleness-factor` is 0.5, the returned strategy cooperates half the time that `strat` defects; if `gentleness-factor` is 1, the returned strategy performs the same as `Patsy`.)

Use `gentle` with a low value for `gentleness-factor` - say, 0.1 - to create two new strategies: `slightly-gentle-Nasty` and `slightly-gentle-Eye-for-Eye`.

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 involve 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 `Eye-for-Eye` 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 `Eye-for-Eye` 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 (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):

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

$$DCC > CCC$$

$$DDD > CDD$$

$$DCD > CCD$$

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:

$$CCD > DDD$$

$$CCC > DCD$$

$$CCD > \frac{CDD + DCD}{2}$$

$$CCC > \frac{CCD + DCC}{2}$$

We can satisfy all of these constraints with the following payoffs:

$CDD = 0, \quad DDD = 1, \quad CCD = 2, \quad DCD = 3, \quad CCC = 4, \quad DCC = 5.$

Problem 9

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` (although you may also have to change your definition of `extract-entry` if you did not write it in a general enough manner). We would suggest that you make copies of the necessary code and rename them so that you can separate the two person version from the three person one.

You also need to change `*game-association-list*` as follows:

```
(define *game-association-list*
  (list (list (list "c" "c" "c") (list 4 4 4))
        (list (list "c" "c" "d") (list 2 2 5))
        (list (list "c" "d" "c") (list 2 5 2))
        (list (list "d" "c" "c") (list 5 2 2))
        (list (list "c" "d" "d") (list 0 3 3))
        (list (list "d" "c" "d") (list 3 0 3))
        (list (list "d" "d" "c") (list 3 3 0))
        (list (list "d" "d" "d") (list 1 1 1))))
```

Problem 10

Write strategies `Patsy-3`, `Nasty-3`, and `spastic-3` that will work in a three-player game. Try them out to make sure your code is working.

Write two new strategies: `tough-Eye-for-Eye` and `soft-Eye-for-Eye`. `Tough-Eye-for-Eye` should defect if *either* of the opponents defected on the previous round. `Soft-Eye-for-Eye` should defect only if `both` opponents defected on the previous round. Play some games using these two new strategies. Describe the observed behavior of the strategies.

Problem 11

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-Eye-for-Eye` in Problem 10.

```
(make-combined-strategies
  Eye-for-Eye Eye-for-Eye
   (lambda (r1 r2) (if (or (string=? r1 "d") (string=? r2 "d"))"d" "c")))
```

The resulting strategy plays `Eye-for-Eye` 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 `Eye-for-Eye` against one opponent, `Egalitarian` against another, and chooses randomly between the two results:

```
(make-combined-strategies
  Eye-for-Eye Egalitarian
  (lambda (r1 r2) (if (= (random 2) 0) r1 r2)))
```

Problem 12

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.

The underlying idea is to keep track of how the strategy for one player correlates with the decisions of the other two players on the previous round (of course, you can imagine generalizing this to several previous rounds). Thus, we want to build an intermediary data structure which keeps track of what player-0 did, correlated with what the other two players did, over the course of the histories for the three players. Imagine creating 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`. Given this is a three player game, there are three possible situations we need to keep track of: what did player-0 do on one round when the two other players both cooperated on the previous round; what did player-0 do on one round when one of the others cooperated and the other defected on the previous round; and what did player-0 do on one round when both other players defected on the previous round. Since these three situations will occur multiple times, we want to keep track of how often in each case did player-0 cooperate, and how often did she defect in response to these choices, and how often did each of these three cases occur (although that could be found by adding the number of times player-0 cooperated and defected).

Thus, you should design and implement a data structure called a `history-summary`, with the overall structure shown in Figure 1. The `history-summary` has three subpieces, one for the case where both player-1 and player-2 cooperated, one for when one of them cooperated and the other defected, and a third for when both of these players defected. This means that your data abstraction for a `history-summary` should have three selectors, for these three pieces. For each piece, there is another data structure that keeps track of the number of times player-0 cooperated on the next round, the number of times she defected, and the total number of examples (though as we noted, this is redundant). You may find it convenient to think of this as a kind of tree structure. Thus, your first task is to design constructors and selectors to implement this multilevel abstraction.

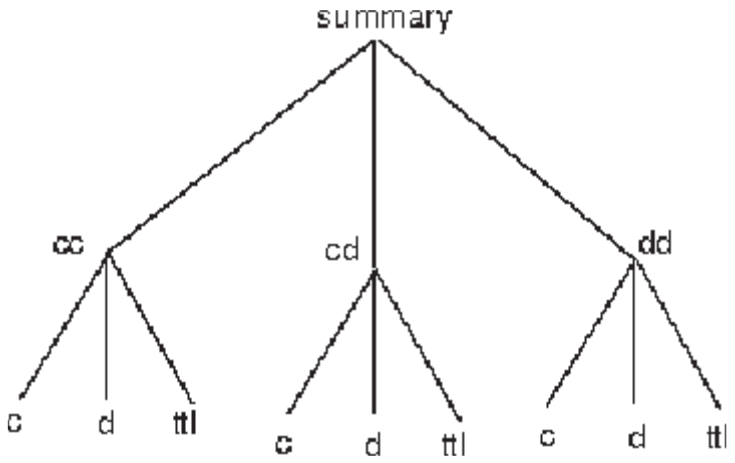

Figure 1: Example of the summary data structure, as a tree. The top level has three pieces, corresponding to the actions of the other players (both cooperated, only one cooperated, both defected). The second level has three pieces, listing the number of times the player cooperated, defected and the total number of times the situation specified by the actions of the opponents occured.

Once you have designed your data abstraction, build a procedure that takes the three histories as arguments, and returns a `history-summary`. If we extract from this data structure the piece corresponding to `cooperate-cooperate`, this should give us all the information about what happened when player-1 and player-2 both cooperated. Thus, we should be able to extract from this piece the number of times player-0 cooperated and the number of times she defected.

REMEMBER: the goal of our data structure is to correlate player-0's behavior on round n, with player-1 and player-2's behavior on round n-1. For example, the result of an implementation, call it `make-history-summary`, on an example set of histories is shown below:

```
(define summary (make-history-summary
                 (list "c" "c" "d" "d" "c" "d" "c" "c")    ;hist-0
                 (list "c" "c" "c" "d" "d" "c" "d" "c")    ;hist-1
                 (list "c" "c" "d" "d" "d" "c" "c" "c")    ;hist-2))


summary
;Value: ((3 0 3) (1 1 2) (0 2 2))
```

To help you decode this result, first remember that since we are going to compare the decision for, say, the most recent round in the first history, this means we compare that value ("c") against the values of the previous round in the other two histories (also both "c"), or we ompare the value in the previous round ("c) against the values in the

preceeding round of the other two histories (a “c” and a “d”). As a result of this process, the first list in this summary describes what player-0 did on a round immediately after both opponents cooperated, in this case she cooperated 3 times, and never defected. The second list describes what player-0 did on a round immediately after one opponent cooperated and one defected, in this case she cooperated once and defected once; and the final list describes what player-0 did on a round immediately after both opponents defected, in this case she defected twice and never cooperated. Note that there are only 7 cases counted, since we compare the result on one round against the opponents’ decisions on the previous round.

Problem 13

Finally, using this data structure, we can build a new procedure that 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.

```
(define summary (make-history-summary
                 (list "c" "c" "c" "c")     ;hist-0
                 (list "d" "d" "d" "c")     ;hist-1
                 (list "d" "d" "c" "c")))   ;hist-2

(get-probability-of-c summary)
;Value: (1 1 1)

(define new-summary (make-history-summary
                      (list "c" "c" "c" "d" "c")
                      (list "d" "c" "d" "d" "c")
                      (list "d" "c" "c" "c" "c")))

(get-probability-of-c new-summary)
;Value: (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 of the other players defect, our procedure returns `()` for that case.

Write the `get-probability-of-c` procedure.

Problem 14

Using this procedure, you should be able to write some predicate procedures that help in deciphering another player's strategy. For instance, we can use `get-probability-of-c` to record the behavior of an opponent. We could then compare this against what we would expect for a behavior to see if they match. Thus, the first procedure tests to see if two lists are the same. Using this we could check to see if an opponent is a fool by seeing if he always cooperates (i.e. the observed behavior would be a "c" for cooperate in all cases).

```
(define (test-entry index trial)
  (cond ((null? index)
         (null? trial))
        ((null? trial) #f)
        ((= (car index) (car trial))
         (test-entry (cdr index) (cdr trial)))
        (else #f)))
```

```
(define (is-he-a-fool? hist0 hist1 hist2)
  (test-entry (list 1 1 1)
              (get-probability-of-c
                 (make-history-summary hist0 hist1 hist2))))
```

```
(define (could-he-be-a-fool? hist0 hist1 hist2)
  (test-entry (list 1 1 1)
              (map (lambda (elt)
                     (cond ((null? elt) 1)
                           ((= elt 1) 1)
                           (else 0)))
                   (get-probability-of-c (make-history-summary hist0
                                                               hist1
                                                               hist2)))))
```

Use the `get-probability-of-c` procedure to write a predicate that tests whether another player is using the `soft-Eye-for-Eye` strategy from Problem 10. 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 `Patsy`. If our strategy finds that both other players seem to be cooperating uniformly, it defects; otherwise, it cooperates.

Submission

For each problem above, include your code (with identification of the problem number being solved), as well as comments and explanations of your code, **and** demonstrate your code's functionality against a set of test cases. Once you have completed this project, your file should be **submitted electronically on the 6.001 on-line tutor**, using the `Submit Project Files` button.

Remember that this is Project 2; when you are have completed all the work and saved it in a file, upload that file and submit it for Project 2.

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

As described earlier, Axelrod held two computer tournaments to investigate the two-player prisoner's dilemma. We are going to hold a three-player tournament. You can participate by designing a strategy for the tournament. You might submit one of the strategies developed in the project (but not one of the standard strategies provided as part of the code), or develop a completely 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. If you wish to submit an entry strategy, you should:

- Send a copy of your procedure by email to your TA by the due date of the project (we will *not* accept entries submitted after the project is due). Include your name and a brief description of how the strategy works.
- The form of the submitted strategy should be a procedure that takes three arguments: the player's own history list and history lists for each of the other two players. The procedure should return either a "c" or a "d" for cooperate or defect.
- We reserve the right to disqualify any entries that violate the spirit of the prisoner's dilemma game (e.g., by "mutating" someone elses's history list).
- We *strongly* suggest that you try out your procedure in the lab (by using it as an argument to the three-person `play-loop` procedure) before submitting it.

The tournament will be a complete one, that is every strategy plays against every other pair. Each game will consist of approximately 100 rounds.