MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2007

Project 2

Release date: Monday, March 5, 2007
Due date: Friday, March 16, at 6pm

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.

Additional guidelines for project submission are available under the "How to write up a project" link off of the course web page.

Read the entire project description below before you start working. **Every sentence in boldface** describes something that you should include in your final handin — e.g., procedures to write, test cases to run, or questions to answer.

This project includes some provided code in the file `mind-read.scm`, which can be obtained from the Projects link on the course web page.

We strongly suggest that you create your own file in which to store your solutions. At the top of this file, use the `load` command, that is, store a copy of `mind-read.scm` in your directory, and then place the expression

```
(load "mind-read.scm")
```

at the top of your solution file. This will cause Scheme to load this file when you evaluate your solution file, and thus incorporate all of the code from that file into your Scheme environment.

Scenario

Ben Bitdiddle has joined the MIT mind-reading club (MMC). Founded in 1892, the club originally concentrated on the game of "paper-rock-scissors", with hundreds of undergraduates playing round after round of this classic game. In 1908, Lem E. Tweakit, an MMC member, in a flash of brilliance invented a game called *true-false* that is similar to paper-rock-scissors, but only involves two outcomes. Since then, MMC has concentrated on true-false as its game of choice.

Lem E. Tweakit's game works as follows. The game consists of two people: the *player* and the *mind-reader*. The *player* gives a series of *labels*, where each label is "true" or "false". At each point, the *mind-reader* tries to predict what the player will say next. Each time the mind-reader predicts correctly, she gets a point. Each time the mind-reader makes a wrong prediction, the player gets a point. The game is played for a number of rounds, and at the end of the game the person with the most points wins.

In general, the mind-reader will win if she can spot a pattern in the player's choices, and then exploit this pattern to make correct predictions. For example, if the last ten choices by the player are (with the most recent one first):

```
#t #f #t #f #t #f #t #f #t #f
```

then it would be reasonable to predict `#f` as the next label from the player. As another example, suppose the player had given the following sequence so far:

```
#t #t #t #f #f #t #t #f #f #f #f #t #t
```

In this case, the mind-reader would hopefully notice that the player has a tendency to repeat the same label (i.e., #t or #f) more than once, and would predict `#t` as the next label. This strategy will not be correct all the time, but it seems it may well be correct more than 50% of the time, which is enough for the mind-reader to win the game.

We will call a sequence of choices made by a player, with the most recent choice first, a **history list**.

Ben Bitdiddle decides to build a program that will play the role of the *mind-reader* in the true-false game. The program will take a series of inputs—true or false—from a player. At each point it will try to predict what the player will say next. At some point the player can choose to end the game, in which case the program will display the score for the player and the mind-reader. Your task in this project will be to write the software for Ben. In particular, our eventual goal will be to develop methods that learn to spot patterns in the player's input, and thereby beat the player on a regular basis.

Ben is excited when he realizes that the software may have applications beyond the true-false game. One application is in modeling the stock market: in this case the sequence of true/false predictions are predictions about whether the market will go up or down the next day. Ben could make a lot of money if he could predict this with any accuracy. Another application is in weather prediction, for example the true/false sequence corresponds to whether or not it rains on a sequence of days. Ben is excited about this application because his friend Alyssa, who is in the other Cambridge as part of the Cambridge-MIT exchange, is having trouble planning barbeques, given the number of days that it rains in the summer.

Part 1: A Simple Mind-Reading Machine

Ben's first attempt at a mind-reading machine is `simple-game`, which is in the code attached with this project. `simple-game` takes a single parameter, `func`, which is a function that takes a list as input, and returns true or false as its output. The list is expected to be a **history list** that is to represent a history (or context) of previous results of the mind-reading game, and is represented with the most recent result at the beginning of the list. The procedure `func` is a strategy for predicting the next choice by a player. For example, the following functions are also included in the code:

```
(define always-true (lambda (x) #t))

(define always-false (lambda (x) #f))
```

```
(define guess-last (lambda (x) (if (null? x) #t (car x))))

(define guess-not-last (lambda (x) (if (null? x) #t (not (car x)))))

(define alternating-true-false (lambda (x) (even? (length x))))
```

Each of these functions takes a list of #t/#f outcomes as input, and returns #t or #f as the output. For example, you can try the following in the read-eval-print loop:

```
(always-true (list #t #f #t #f)) ;-> #t
(always-false (list #t #f #t #f)) ;-> #f
(guess-last (list #t #f #t #f)) ;-> #t
(guess-not-last (list #t #f #t #f)) ;-> #f
(alternating-true-false (list #t #f #t #f)) ;-> #t
```

where we are using the notation `;->` to indicate that value to which the expression evaluates.

Try playing against the game by loading the code into DrScheme, and typing

```
(simple-game always-true)
```

to the read-eval-print loop. This will start a game where you play the role of the *player*, and the code plays the role of the *mind-reader*. At each round you can enter `t` (for true), `f` (for false), or `e` (to end the game). In addition, the code makes a prediction by applying the function `func` (in this case `always-true`) to the history of your previous predictions. The code keeps a running total of the score for the player and the reader, as well as the history of previous predictions by the player.

Make sure that you understand the code in `simple-game`. Try playing `simple-game` for the functions `always-true`, `always-false`, `guess-last`, `guess-not-last` and `alternating-true-false`.

Question 1 : In each of these cases, find a sequence of 10 inputs by the player that results in a final score of 10 for the player and 0 for the mind-reader on a game of 10 rounds. You should include these sequences as part of what you submit for your project.

Note: `simple-game` is using a very simple strategy, where the mind-reader consistently predicts the next output by applying the function `func` to the history of previous predictions by the player. This is clearly a very naive strategy, at least for the five functions listed above, and it's easy to beat! One goal of this project will be to develop algorithms that do much better at predicting the person's actions at each point.

Part 2: Creating More Functions

Remember that Ben's goal is to build a machine that will read a player's mind. Clearly `simple-game` is not sufficient, as it is easy (as you discovered) for a player to beat the machine. Ben realizes that the functions `always-true` to `alternating-true-false` listed above aren't going to be good enough if he wants to do well at the mind-reading game. He decides to create some new functions.

The goal is to create more sophisticated functions whose behavior is harder to predict simply by observing their output.

We are going to do this in stages, by first creating some functions that are more sophisticated in the information they use from previous choices to predict the next outcome, and then combining those functions into higher order functions that merge behavior.

First, we'll define a function `(skip-most-recent f)` which takes a function `f` as its input. As its output, it should return a new function `g` which has the following behavior:

$$\begin{aligned} g(x) &= \text{\#t if } x \text{ is the empty list or a list of length 1} \\ &= f(y) \text{ where } y = \text{(cdr x) otherwise} \end{aligned}$$

For example, the function should have the following behavior:

```
((skip-most-recent guess-last) (list #t #f #t #f)) ;-> #f
```

because this is equivalent to

```
(guess-last (list #f #t #f))
```

(Remember that a history list has the most recent choice first in the list.)

Question 2 : Define `skip-most-recent` as a function of the following form:

```
(define (skip-most-recent f)
   YOUR-CODE-HERE)
```

Play `simple-game` using `skip-most-recent` applied to one of the base functions.

Question 3 : We can generalize this to create a function that "looks back" some arbitrary number of times. Create a procedure `skip-most-recent-n` that takes two arguments, an integer `n` and a function `f`

```
(define (skip-most-recent-n n f)
   YOUR-CODE-HERE)
```

It should return a new function `g` which behaves as follows:

$$\begin{aligned} g(x) &= \text{\#t if } x \text{ is a list of length less } n \\ &= f(y) \text{ where } y = \text{the rest of list } x \text{ after skipping the first } n \text{ elements, otherwise} \end{aligned}$$

Thus, `(skip-most-recent-n 1 f)` should behave the same as `(skip-most-recent f)`.

Note that there are many ways of writing `skip-most-recent-n`. One way is to modify the history list of previous choices, and then simply apply the function to that list. A second way, and **the one we want you to use here**, is to use `skip-most-recent`, that is, if the argument `n` is 1, `skip-most-recent-n` should apply `skip-most-recent`; if it is greater than 1, then `skip-most-recent-n` should recursively call itself on an appropriate new function.

Try running `simple-game` on some version of `skip-most-recent-n` applied to `guess-last`. Describe the strategy you need in order to get a perfect score against this function.

Next, we'll define a function `(negation f)` which takes a function `f` as its input. As its output, it should return a new function `g` which has the following behaviour:

$$\begin{aligned} g(x) &= \#\text{t if } f(x) = \#f \\ &= \#\text{f if } f(x) = \#t \end{aligned}$$

Question 4 : Define `negation` as a function of the following form:

```
(define (negation f)
   YOUR-CODE-HERE)
```

Another strategy is to look for sequences of consistent choices, for example, to have a function that returns #t if the previous n choices were #t.

Question 5 : Define `n-in-a-row` as a function of the following form:

```
(define (n-in-a-row n)
   YOUR-CODE-HERE)
```

which returns a procedure of one argument that can be applied to a list of prior choices. `n-in-a-row` should return #t if its argument has fewer than n elements, should return #f if the next choice was #f, and otherwise should recursively call itself on the rest of the list of choices, unless n is equal to 1, in which case it should return the value of the next element.

We can also define a function `(fand f1 f2)` which takes two functions `f1` and `f2` as its input. As its output, it should return a new function `g` which has the following behaviour:

$$\begin{aligned} g(x) &= \#\text{t if } f1(x) = \#t \text{ and } f2(x) = \#t \\ &= \#\text{f otherwise} \end{aligned}$$

Question 6 : Define `fand` as a function of the following form:

```
(define (fand f1 f2)
   YOUR-CODE-HERE)
```

Finally, we can now combine strategies together. For example, using `fand` we can write a strategy that will return #t if the last choice was #t, but the previous three choices were not all #t.

Question 7 : Write and test a strategy, `true-but-not-always-true`, with the following behavior:

- **it returns #t if the last choice was true, but the last three choices were not all true,**
- **it returns #f if there have been fewer than three previous choices,**
- **otherwise it returns #f**

Be sure to show examples testing your code in each of the above questions. Use these functions to create and test some new function, using `simple-game`.

Part 3: Generating Sequences

Ben decides that it would be good if he also had code to *generate* sequences of choices, so that he can test his different strategies for a mind-reader. This will enable him to carefully explore the behavior of functions he writes on specifically designed test sequences of inputs. He would like a function `(generate func n)` which takes a function `func`, and a positive integer `n` as input. As in our other examples, the function `func` should map a list of #t/#f predictions to #t or #f. `generate` will generate a sequence of length `n` by repeatedly applying `func` `n` times (that is first to the empty list, then to the list containing the first response, then to the list containing the first two responses, and so on). In other words, `generate` is creating a sequence of the first n choices made by `func`, given its own previous results as a history at each step. The order of the sequence should have the first response as the FIRST element of the list or sequence. Be careful here – this FUTURE LIST is different from a history list, as it contains elements in the opposite order from a history list. We are doing this because we want to use a simple batch game mode (described below) to test our strategies, and that code (as you can see) treats the results of `generate` as a FUTURE LIST of choices as if they were made by a player.

Thus, our procedure should have the behavior:

```
(generate always-true 10) ;-> (#t #t #t #t #t #t #t #t #t #t)
(generate always-false 10) ;-> (#f #f #f #f #f #f #f #f #f #f)
(generate guess-not-last 10) ;-> (#t #f #t #f #t #f #t #f #t #f)
```

<u>Question 8</u> : Write a function `generate` which has the form shown below:

```
(define (generate func n) YOUR-CODE-HERE)
```

Test your function on the following test cases:

```
(generate (skip-most-recent guess-not-last) 20)
(generate (skip-most-recent (skip-most-recent guess-not-last)) 20)
(generate (skip-most-recent-n 1 guess-not-last) 20)
(generate (skip-most-recent-n 2 guess-not-last) 20)
```

Note: to help you test your code, we've also provided a function, `(batch-simple-game sequence func)` which has the following behavior. You can call `batch-simple-game` with a specific sequence of #t/#f predictions, and a function. For example, try

```
(batch-simple-game (list #t #t #t #t) always-true)
```

In this case `batch-simple-game` will return as its value the number of errors when `always-true` is used to play on the input sequence `(list #t #t #t #t)` from the user. In this case the value returned will be 0, because `always-true` perfectly predicts every label given in the sequence. Note that in general, for any function `f` we should have

```
(batch-simple-game (generate f 10) f)
```

return the value 0, because `f` will be a perfect predictor on a sequence generated by itself. So `batch-simple-game` provides a way of testing a function on specific sequences.

NOTE: if you are still uncertain about the order in which `generate` should create sequences, look carefully at the code for `batch-simple-game` to see how it uses each element of the resulting list as a choice.

Question 9 : Design some test cases for `generate`. For example, you can compare the output of `generate` to the expected result for simple strategies. You can also use `batch-simple-game` to compare two strategies, for example:

```
(batch-simple-game (generate (negation guess-not-last) 10)
                   guess-not-last)
```

should return the value 10, since these two strategies should always disagree. Explain the behavior for each of these cases.

Part 4: A Random Method

Ben now decides to design a randomized function. The function takes the form `(random-choice f1 f2 p)`. The arguments `f1` and `f2` are both functions. `p` is a value between 0 and 1. `random-choice` should return a new function that has the following behavior: for any input x, with probability p it should return `(f1 x)`, and with probability $1 - p$ it should return `(f2 x)`. For example, if we try

```
(generate (random-choice f1 f2 0.5) 20)
```

then `generate` should randomly choose between functions `f1` and `f2` at each point when generating the sequence. The result should be a random true-false sequence where there are a roughly equal number of true's and false's.

Ben's first attempt at the function looks like the following:

```
(define (bad-random-choice f1 f2 p)
   (let ((q (random-fraction)))
      (if (<= q p)
          (lambda (x) (f1 x))
          (lambda (x) (f2 x)))))
```

(Note: the function `random-fraction` returns a random value between 0 and 1.) As its name suggests, this function does not behave as Ben hoped it would. Try running

```
(generate (bad-random-choice always-true always-false 0.5) 10)
```

a few times in the read-eval-print loop.

Question 10 : What is the problem here?

Question 11 : Define and test your own version of `random-choice` that has the correct behavior:

```
(define (random-choice f1 f2 p)
   YOUR-CODE-HERE)
```

Question 12 : We want you to design a new strategy, with the following behavior. This strategy with probability 1/2 should act like `guess-last`, with probability 1/4 should act like `(skip-most-recent guess-last)`, with probability 1/8 should act like `(skip-most-recent (skip-most-recent guess-last))`, and in general should probability $\frac{1}{2^n}$ choose the nth previous result. You will want to use `random-choice` as part of your solution. Test it using `batch-simple-game`.

Part 5: Looking at More of the History

Ben realises that a particularly good strategy might be to look at the number of times that the player has chosen `#t` or `#f` in the recent history, and make a prediction based on these counts. He decides to implement a function `(lastn n)` which works as follows. The argument `n` is assumed to be an integer that is greater than 0. `(lastn n)` should return a function g which takes a list x as input and has the following behavior:

$$\begin{aligned} g(x) &= \text{\#t if \#t is seen at least as many times as \#f} \\ & \quad \text{in the first } n \text{ members of } x \\ &= \text{\#f otherwise} \end{aligned}$$

For example, you should have

```
((lastn 3) (list #t #t #f #f)) ;-> #t
((lastn 3) (list #t #f #f #f)) ;-> #f
```

Note: if the length of the input list x is less than n, then the function returned by `(lastn n)` should look at the number of times #t and #f appear in the list x, ignoring the fact that x has fewer than n members. For example:

```
((lastn 3) (list #t #t)) ;-> #t
((lastn 3) (list #t)) ;-> #t
((lastn 3) '()) ;-> #t
```

Question 13 : Implement and test a version of `lastn` which has this behavior:

(define (lastn n) YOUR-CODE-HERE)

Part 6: A Learning Algorithm

Ben now decides to build a far more powerful mind-reading machine. His main idea is to build a program that learns from the player's previous inputs, in attempting to make the correct prediction at each step.

We will build a function, `(learning-game funcs)` which takes a list of functions as its only input. For example, we might define `funcs` to be the list of eight functions:

```
(define funcs
  (list always-true
        always-false
        guess-last
        guess-not-last
        alternating-true-false
        (skip-most-recent guess-last)
        (skip-most-recent guess-not-last)
        (skip-most-recent alternating-true-false)))
```

and then run `learning-game` with this list of functions:

```
(learning-game funcs)
```

We've included code for `learning-game` in the code attached to the project. It is missing definitions for a couple of functions, namely `prediction` and `update-weights`, which we will come to shortly. Once these functions have been implemented, `learning-game` will behave in a very similar way to `simple-game`, which you saw earlier. At each round it will ask the player to input `t` (for `#t`) or `f` (for `#f`). At each round it will also make a prediction of what the player is going to input. It will keep track of the score of the player and the mind-reader, as well as the previous history of inputs from the player. When the player enters `e`, to end the game, it will display the scores for the player and the mind-reader.

As we've seen, a major difference between `learning-game` and `simple-game` is that `learning-game` takes a list of functions as input, whereas `simple-game` takes a single function. We now describe how `learning-game` uses the entire list of functions to make its prediction at each step.

A key property of `(learning-game funcs)` is that it maintains a list of *weights*, one for each of the functions in the list `funcs`. Initially, these weights are all set to be the same value, i.e., 1. Given a list of weights, and a list of functions, `learning-game` makes a prediction by taking a *weighted vote* of the output of the different functions. Each function "votes" for its output (true or false) with its weight. The overall prediction, true or false, depends on which of the two outputs gets the highest weight.

To illustrate this, consider the following example. We will call the function that takes the weighted vote of a list of functions `(prediction funcs weights ctxt)`. Here `funcs` is a list of functions, `weights` is a list of weights for the functions, and `ctxt` is a history to which the functions will be applied. An example usage would be

```
(prediction (list always-true always-false guess-last guess-not-last)
            (list 0.1 0.2 0.3 0.7)
            (list #t #f #t #f))
```

To see what the output should be in this case, first let's calculate the prediction of each of the four functions on the history:

```
(always-true (list #t #f #t #f)) ;-> #t
(always-false (list #t #f #t #f)) ;-> #f
(guess-last (list #t #f #t #f)) ;-> #t
(guess-not-last (list #t #f #t #f)) ;-> #f
```

These four functions get to vote with weights 0.1, 0.2, 0.3 and 0.7 respectively (see the list above). Thus #t gets a total vote of $0.1 + 0.3 = 0.4$; #f gets a total vote of $0.2 + 0.7 = 0.9$. In this case #f gets the highest total vote, so this is the final output from `prediction`.

Question 14 : Write the function `prediction` which has this behavior: (Note: in the case where the total weighted votes for #t and #f are equal (a tie), your function should output #t.)

```
(define (prediction funcs weights history)
  YOUR-CODE-HERE)
```

Note: you may find the following function useful:

```
(define (calc-votes funcs history)
   (map (lambda (f) (f history)) funcs))
```

This function takes a list of functions, and a history, and returns a list of the predictions of each of the functions on this history. For example, try

```
(calc-votes (list always-true always-false guess-last guess-not-last)
            (list #t #f #t #f))
(calc-votes (list always-true always-false) (list #t #f #t #f))
```

Be sure to show your tests of this function.

Part 7: Implementing `update-weights`

Now we've implemented `prediction`, the remaining task is to implement the function

```
     (update-weights funcs weights history actual-choice).
```

As in `prediction`, this function has the arguments `funcs`, `weights` and `history` which are respectively a list of functions, a list of weights, and a history. In addition, `actual-choice` is an argument that specifies the actual input given by the *player*, either #t or #f. `update-weights` gives a new list of weights as its return value.

`update-weights` calculates the new weights as follows:

- As a first step, it calculates the value for each of the functions in `funcs` when applied to the `history`.
- For any function which correctly predicts `actual-choice` as its output, its weight remains the same.
- For any function which does **not** predict `actual-choice` as its output, its weight is multiplied by a factor of 0.3.

As an example, consider `update-weights` applied to the following arguments:

```
(update-weights (list always-true always-false guess-last guess-not-last)
                (list 0.1 0.2 0.3 0.7)
                (list #t #f #t #f)
                #t)
```

In this example, functions `always-true` and `guess-last` correctly predict `#t` when applied to the history `(list #t #f #t #f)`; the weights for `always-true` and `guess-last` therefore remain the same. In contrast, functions `always-false` and `guess-not-last` make the incorrect prediction of `#f`. The weights for these two functions are multiplied by a factor of 0.3. The return value for `update-weights` is then

```
(0.1 0.06 0.3 0.21)
```

You should see that there is a clear intuition behind this behavior: each time a function makes an error, its weight is decreased by a factor of 0.3, meaning that it will have less influence in future rounds of voting. In particular, if `update-weights` is called round by round of `learning-game`, functions which make a large number of errors will have their weight quickly decrease towards 0.

Question 15 : Write the function `update-weights` which has this behavior:

```
(define (update-weights funcs weights history actual-choice)
  YOUR-CODE-HERE)
```

Part 8: Testing the Learning Game

We're now ready to test the code in `learning-game`. First, you should define `allfuncs` as the following (we have this definition commented out in the code attached to the project, you should copy it to your file, and remove the ; so that evaluating your file will now evaluate this expression):

```
(define allfuncs
  (list always-true always-false guess-last guess-not-last
        (skip-most-recent always-true) (skip-most-recent always-false)
        (skip-most-recent guess-last) (skip-most-recent guess-not-last)
        (skip-most-recent (skip-most-recent always-true))
        (skip-most-recent (skip-most-recent always-false))
```

```
          (skip-most-recent (skip-most-recent guess-last))
          (skip-most-recent (skip-most-recent guess-not-last))
          (lastn 3) (lastn 5) (lastn 7)
          (negation (lastn 3)) (negation (lastn 5)) (negation (lastn 7))))
```

Question 16 : Next, try playing against the game by typing

```
(learning-game allfuncs)
```

Try playing for around 100 rounds. Could you beat the game? Did it beat you?

We've also provided a non-interactive version of `learning-game`, the function `(batch-learning-game sequence funcs)`. This function takes two arguments: first, a list of #t/#f predictions; second, a list of functions that will be used by the game. `batch-learning-game` applies the learning method to the `sequence`, using the given `funcs`. For example, you can try

```
(batch-learning-game (list #t #t #t #t) allfuncs)
```

Question 17 : Try `batch-learning-game` on the following sequences:

```
(batch-learning-game (generate always-true 100) allfuncs)
(batch-learning-game (generate always-false 100) allfuncs)
(batch-learning-game (generate guess-not-last 100) allfuncs)
(batch-learning-game (generate (skip-most-recent guess-not-last) 100) allfuncs)
(batch-learning-game (generate (skip-most-recent
                                  (skip-most-recent guess-not-last)) 100)
                     allfuncs)
(batch-learning-game (generate (random-choice always-true always-false 0.5) 100)
                     allfuncs)
(batch-learning-game (generate (random-choice always-true always-false 0.8) 100)
                     allfuncs)
```

In each case state how many errors `batch-learning-game` made, and give an explanation of this behavior.

Part 9: Creating a New Function

We've seen that `allfuncs` makes use of a number of functions: `always-true, always-false, guess-last, guess-not-last, (lastn 3), (lastn 5)`, and so on. We've chosen these functions to try to do as well as possible when `learning-game` is played against a human.

Question 18 : Design one or more of your own functions, which you think will be good additions to `allfuncs`, when `allfuncs` is used to play against a human player. In each case describe the behavior of the function, and motivate why you think it will be a useful addition to `allfuncs`.

Part 10: Creating a New Sequence

We've seen that `batch-learning-game` performs pretty well against a variety of sequences that are generated using `generate`. In this question your task is to design a sequence that results in a large number of errors for `batch-learning-game`.

Question 19 : Design a new function

```
(define (winning-sequence n)
  YOUR-CODE-HERE)
```

which takes an integer n as its one argument, and returns a sequence of #t/#f decisions of length n as its value. Once you've done this, test `batch-learning-game` against a sequence of length 100:

```
(batch-learning-game (winning-sequence 100) allfuncs)
```

You should: 1) give your definition of `winning-sequence`; 2) describe its intended behavior; 3) state the number of errors that `batch-learning-game` makes when playing against it.

A few notes on this:

- Your goal in this question is to cause `batch-learning-game` to make as many errors as possible on your sequence.
- Your function must be **deterministic**: this means it is not allowed to make use of random choices, for example by calls to the procedures `random-fraction` or `random`.
- There are a variety of ways of designing `winning-sequence`. One would be to define a new function, `f`, which maps true/false sequences to true or false (similar to `always-true, always-false, guess-not-last`, and so on), and then use this to generate a sequence:

  ```
  (define (winning-sequence n)
     (generate f n))
  ```

  Another method is to first compute some irrational number, for example `e`, and then to use the binary expansion of this number as your sequence. With the right choice of irrational number the resulting sequence will be close to random, and will perform well against `batch-learning-game`.

  Extra credit: You may even be able to come up with a method that causes `batch-learning-game` to make 100 errors in 100 rounds! If you do manage to do this, how would you alter `batch-learning-game` so that it performs ok even against this function? (Hint: you can use randomization within `batch-learning-game`.)