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.

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 predictions, and then exploit this pattern to make correct predictions. For example, if the last ten predictions by the player are

#t #f #t #f #t #f #t #f #t

then it would be reasonable to predict #t 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

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.

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, still in England, is having trouble planning barbecues, given the number of days that it rains in the summer.

**Problem 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. For example, the following four functions are also included in the code:

```scheme
(define f1 (lambda (x) #t))
(define f2 (lambda (x) #f))
(define f3 (lambda (x) (if (null? x) #t (car x)))))
(define f4 (lambda (x) (if (null? x) #t (not (car 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:

```
(f1 (list #t #f #t #f)) ;→ #t  
(f2 (list #t #f #t #f)) ;→ #f  
(f3 (list #t #f #t #f)) ;→ #t  
(f4 (list #t #f #t #f)) ;→ #f
```

where we are using the notation ;→ to indicate that value to which the expressions evaluates.

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

```
(simple-game f1)
```

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 f1) 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 \texttt{f1}, \texttt{f2}, \texttt{f3}, and \texttt{f4}.

**Question 1**: In each case, 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 \texttt{func} to the history of previous predictions by the player. This is clearly a very naive strategy, at least for the four 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.

**Problem 2: Creating More Functions**

As a first step in improving upon simple-game, Ben realizes that the functions \texttt{f1} to \texttt{f4} 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.

First, we’ll define a function \texttt{(twoback \ f)} which takes a function \texttt{f} as its input. As its output, it should return a new function \texttt{g} which has the following behavior:

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

For example, the function should have the following behavior:

\[
((\text{twoback } f3) \ (\text{list } \#t \ #f \ #t \ #f)) \rightarrow \#f
\]

because this is equivalent to

\[
(f3 \ (\text{list } \#f \ #t \ #f))
\]

**Question 2**: Define \texttt{twoback} as a function of the following form:

\[
\text{(define (twoback f)}
\text{\hspace{1cm} YOUR-CODE-HERE})
\]

**Question 3**: We can generalize this to create a function that “looks back” some arbitrary number of times. Create a procedure \texttt{nback} that takes two arguments, an integer \texttt{n} and a function \texttt{f}. It should return a new function \texttt{g} which behaves as follows:

\[
g(x) = \begin{cases} 
\#t & \text{if } x \text{ is a list of length less } n \\
\quad = f(y) & \text{where } y = \text{the } n^{th} \text{ element of } x \text{ otherwise}
\end{cases}
\]
Thus, \((\text{nback 2 } f)\) should behave the same as \((\text{twoback } f)\).

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

\[
g(x) = \begin{cases} #\text{t} & \text{if } f(x) = #\text{f} \\ #\text{f} & \text{if } f(x) = #\text{t} \end{cases}
\]

**Question 4**: Define \(\text{negation}\) as a function of the following form:

\[
(\text{define (negation } f) \\
\text{YOUR-CODE-HERE})
\]

Finally, we’ll define a function \((\text{fand } f1 f2)\) 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:

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

**Question 5**: Define \(\text{fand}\) as a function of the following form:

\[
(\text{define (fand } f1 f2) \\
\text{YOUR-CODE-HERE})
\]

Be sure to show examples testing your code in each of the above questions.

**Problem 3: Generating Sequences**

Ben decides that it would be good if he also had code to generate sequences, so that he can test his code on various sequences. 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 \((\text{generate } func n)\) which takes a function \(\text{func}\), and a positive integer \(n\) as input. As in our other examples, the function \(\text{func}\) should map a list of \#t/#f predictions to \#t or \#f. \(\text{generate}\) will generate a sequence of length \(n\) by repeatedly applying the function \(n\) 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). The order of the sequence should have the first response as the first element of the list or sequence.

For example, it should have the behavior

\[
(\text{generate } f1 10) \rightarrow (#\text{t} #\text{t} #\text{t} #\text{t} #\text{t} #\text{t} #\text{t} #\text{t} #\text{t}) \\
(\text{generate } f2 10) \rightarrow (#\text{f} #\text{f} #\text{f} #\text{f} #\text{f} #\text{f} #\text{f} #\text{f} #\text{f}) \\
(\text{generate } f4 10) \rightarrow (#\text{f} #\text{t} #\text{f} #\text{t} #\text{t} #\text{f} #\text{t} #\text{f} #\text{f})
\]

**Question 6**: Write a function \(\text{generate}\) which has the form shown below:
(define (generate func n) YOUR-CODE-HERE)

Test your function on the following test cases:

(generate (twoback f4) 20)
(generate (twoback (twoback f4)) 20)
(generate (nback 1 f4) 20)
(generate (nback 2 f4) 20)

Note: to help you test your code, we’ve also provided a function, (simple-game2 sequence func) which has the following behavior. You can call simple-game2 with a specific sequence of #t/#f predictions, and a function. For example, try

(simple-game2 (list #t #t #t #t) f1)

In this case simple-game2 will return as its value the number of errors when f1 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 f1 perfectly predicts every label given in the sequence. Note that in general, for any function f we should have

(simple-game2 (generate f 10) f)

return the value 0, because f will be a perfect predictor on a sequence generated by itself. So simple-game2 provides a way of testing a function on specific sequences.

Question 7: design some test cases for generate, using simple-game2. Explain the behavior for each of these cases.

Problem 4: A Random Method

Ben now decides to design a randomized function. The function takes the form (tworand f1 f2 p). The arguments f1 and f2 are both functions. p is a value between 0 and 1. tworand 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 (tworand 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 (badtworand f1 f2 p)
  (let ((q (our-random)))
    (if (<= p q)
        (lambda (x) (f1 x))
        (lambda (x) (f2 x)))))

(Note: as in the previous project, our-random 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 (badtworand f1 f2 0.5) 10)

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

**Question 8**: What is the problem here?

**Question 9**: Define and test your own version of tworand that has the correct behavior:

(define (tworand f1 f2 p)
  YOUR-CODE-HERE)

**Problem 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 predicted #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. (last n) should return a function g which takes a list x as input and has the following behavior:

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

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 10**: Implement and test a version of `lastn` which has this behavior:

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

**Problem 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 six functions:

```
(define funcs (list f1 f2 f3 f4 (twoback f3) (twoback f4)))
```

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 f1 f2 f3 f4) (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:
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 11**: 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.)

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

Note: you may find the following function useful:

```scheme
(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

```scheme
(calc-votes (list f1 f2 f3 f4) (list #t #f #t #f))
(calc-votes (list f1 f2) (list #t #f #t #f))
```

**Problem 7: Implementing update-weights**

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

```scheme
(update-weights funcs weights history truth).
```

As in `prediction`, this function has the arguments `funcs`, `weights` and `history` which respectively a list of functions, a list of weights, and a history. In addition, `truth` is an argument that specifies the “true” label—i.e., the 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 `truth` as its output, its weight remains the same.
For any function which does not predict truth 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 f1 f2 f3 f4)
   (list 0.1 0.2 0.3 0.7)
   (list #t #f #t #f)
   #t)
```

In this example, functions `f1` and `f3` correctly predict `#t` when applied to the history `(list #t #f #t #f)`; the weights for `f1` and `f3` therefore remain the same. In contrast, functions `f2` and `f4` 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 12**: Write the function `update-weights` which has this behavior:

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

**Problem 8: Testing the Learning Game**

We’re now ready to test the code in `simple-game`. First, you should define `allfuncs` as the following (we have this definition commented out in the code attached to the project, you can just remove the ; and evaluate):

```
(define allfuncs (list f1 f2 f3 f4
   (twoback f1) (twoback f2)
   (twoback f3) (twoback f4)
   (twoback (twoback f1)) (twoback (twoback f2))
   (twoback (twoback f3)) (twoback (twoback f4))
   (lastn 3) (lastn 5) (lastn 7)
   (negation (lastn 3)) (negation (lastn 5))
   (negation (lastn 7))))
```

**Question 13**: 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, which is the function (learning-game2 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. learning-game2 applies the learning method to the sequence, using the given funcs. For example, you can try

(learning-game2 (list #t #t #t #t) allfuncs)

**Question 14**: Try learning-game2 on the following sequences:

(learning-game2 (generate f1 100) allfuncs)
(learning-game2 (generate f2 100) allfuncs)
(learning-game2 (generate f4 100) allfuncs)
(learning-game2 (generate (twoback f4) 100) allfuncs)
(learning-game2 (generate (twoback (twoback f4)) 100) allfuncs)
(learning-game2 (generate (tworand f1 f2 0.5) 100) allfuncs)
(learning-game2 (generate (tworand f1 f2 0.8) 100) allfuncs)

In each case state how many errors learning-game2 made, and give an explanation of this behavior.

**Problem 9: Creating a New Function**

We’ve seen that allfuncs makes use of a number of functions: f1, f2, f3, f4, (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 15**: 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.

**Problem 10: Creating a New Sequence**

We’ve seen that learning-game2 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 learning-game2.

**Question 16**: Design a new function

(define (mysequence 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 learning-game2 against a sequence of length 100:

\[
(\text{learning-game2} \ (\text{mysequence} \ 100) \ \text{allfuncs})
\]

You should: 1) give your definition of \( \text{mysequence} \); 2) describe its intended behavior; 3) state the number of errors that \( \text{learning-game2} \) makes when playing against it.

A few notes on this:

- Your goal in this question is to cause \( \text{learning-game2} \) 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 \( \text{our-random} \) or \( \text{random} \).
- There are a variety of ways of designing \( \text{mysequence} \). One would be to define a new function, \( f \), which maps true/false sequences to true or false (similar to \( f_1, f_2, f_4 \), and so on), and then use this to generate a sequence:

\[
(\text{define} \ (\text{mysequence} \ n) \\
 (\text{generate} \ f \ n))
\]

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 \( \text{learning-game2} \).

Extra credit: You may even be able to come up with a method that causes \( \text{learning-game2} \) to make 100 errors in 100 rounds! If you do manage to do this, how would you alter \( \text{learning-game2} \) so that it performs ok even against this function? (Hint: you can use randomization within \( \text{learning-game2} \).)