University of Waterloo

Megha Bhatt

Contents

1 Introduction

This document was created for a Computer Science project using Python and Dr. Scheme. It is written in Latex, a document markup language. The programs, along with their desgin recipes, are created to do several tasks, which include listing top movies, radix sorting method, listing negative standing students and function of the game of Hangman. The document consists of a table of contents for reference.

2 Python Codes

2.1 Radix Sorting Method

The following code is written in Python. It produces a function that sorts a list using radix sorting method

```
import check

## Constants
buckets = [[],[],[],[],[],[],[],[],[],[]]

## bucket_assigner: int[>=0] int[>0] -> (listof (listof int(>=0)))
## Purpose: Consumes a non-negative integer and a natural number
## and produces a list of lists, where the integer will be placed
## in exactly one of the lists.
def bucket_assigner(number, position):
    pos_num = (number % position) / (position / 10)
    if pos_num == 0:
        buckets[0] = buckets[0] + [number]
        return buckets
    elif pos_num == 1:
        buckets[1] = buckets[1] + [number]
        return buckets
    elif pos_num == 2:
        buckets[2] = buckets[2] + [number]
        return buckets
    elif pos_num == 3:
        buckets[3] = buckets[3] + [number]
        return buckets
    elif pos_num == 4:
        buckets[4] = buckets[4] + [number]
        return buckets
    elif pos_num == 5:
        buckets[5] = buckets[5] + [number]
        return buckets
    elif pos_num == 6:
        buckets[6] = buckets[6] + [number]
        return buckets
    elif pos_num == 7:
        buckets[7] = buckets[7] + [number]
        return buckets
    elif pos_num == 8:
        buckets[8] = buckets[8] + [number]
        return buckets
    else:
        buckets[9] = buckets[9] + [number]
        return buckets
```

```
## buckassign_mapper: (listof int[>=0]) int[>0] (listof (list int[>=0]))
##                       -> (listof (list int[>=0]))
## Purpose: Consumes a list of non-negative integers, a positive integer
## called position, and a list of lists containing non-negative integers.
## Assigns each of the integers in the list of integers to one list within
## the list of lists, and produces the list of lists of integers.
def buckassign_mapper(listofnum, position, bucklist):
    if listof num == []:
        return bucklist
    else:
        return buckassign_mapper(listofnum[1:],\
                                 position,\
                                 bucket_assigner(listofnum[0],\
                                                 position))

## bucket_reducer: (listof (listof int[>=0])) -> (listof int[>=0])
## Purpose: Consumes a list of lists of non-negative integers and folds
## them so that in the produced list of non-negative integers, the
## integers are in the order they appear in the consumed list
def bucket_reducer(listofbuckets):
    lst = reduce (lambda x, y: x + y, listofbuckets)
    buckets[0] = []
    buckets[1] = []
    buckets[2] = []
    buckets[3] = []
    buckets[4] = []
    buckets[5] = []
    buckets[6] = []
    buckets[7] = []
    buckets[8] = []
    buckets[9] = []
    return lst

## radix_main_helper: (listof int[>=0]) int[>=0] int[>=0]
##                       -> (listof int[>=0])
## Purpose: Consumes a list of non-negative integers and two positive
## integers and produces the list sorted using radix sorting method
def radix_main_helper(lst, factor, counter):
    multiplier = 10 ** counter
    if counter == factor + 1:
        return lst
    else:
        return radix_main_helper\
                (bucket_reducer\
                 (buckassign_mapper(lst, multiplier, buckets)),\
                 factor, counter + 1)
```

```
## radix_sort: (listof int[>=0]) -> (listof int[>=0])
## Purpose: Consumes a list of non-negative integers and produces this list
## sorted, using the radix sort method
## Examples:
## radix_sort([0]) => 0
## radix_sort([342]) => [342]
## radix_sort([2540, 39, 173, 5, 60]) => [5, 39, 60, 173, 2540]
## radix_sort([24356, 87, 374, 248735, 4]) => [4, 87, 374, 25356, 248735]
## radix_sort([374832974, 9]) => [9, 374832974]

def radix_sort(L):
    maximum = str(max(L))
    length = len(maximum)
    return radix_main_helper(L, length, 1)

# Tests:
check.expect("ZeroT", radix_sort([234, 0, 19]), [0, 19, 234])

check.expect("OneList", radix_sort([39]), [39])

check.expect("AllDigT", radix_sort([248, 35, 10, 67, 99, 1234567890]),\
             [10, 35, 67, 99, 248, 1234567890])

check.expect("LongNumT", radix_sort([9281748127498127491, 76, 29137]),\
             [76, 29137, 9281748127498127491])
```

2.2 Listing Top Movies

The following code is written in Python. It produces the list of movie objects that occur most frequently in a given list of years.

```
import check

class Movie:
    '''The Movie class has 4 fields:
        rank - a positive integer giving the overall rank of the movie in the
               database. (note that each movie has a unique rank)
        rating - a floating point number between 0.0 and 10.0 giving the overall
                 rating assigned to this movie by the visitors to imdb.com
                 (higher ranked movies have higher ratings)
        title - a non-empty string giving the title of the movie
        year - an integer between 1900 and 2012, for the year the movie was
               released'''

    # __init__: int[>0] float[0.0:10.0] str[len>0] int[1900:2012] -> Movie
    # Purpose: Constructor for class Movie
    # Note: Function definition needs a self parameter and does not require a
    #       return statement
    # Example: Movie(188, 8.0, "The Princess Bride", 1987) creates a Movie object
    # with the rank field set to 188, the rating field set to 8.0, the title field
    # set to "The Princess Bride", and the year field set to 1987.

    def __init__(self, the_rank, the_rating, the_title, the_year):
        self.rank = the_rank
        self.rating = the_rating
        self.title = the_title
        self.year = the_year

    # __repr__: Movie -> str
    # Purpose: Produces a string representation of a Movie
    # Usage: print Movie(188, 8.0, "The Princess Bride", 1987), will print the line
    # 1987: The Princess Bride (rating 8.0, overall rank 188)

    def __repr__(self):
        return "%d: %s (rating %.1f, overall rank %d)" % \
                (self.year, self.title, self.rating, self.rank)

    # __eq__: Movie Movie -> bool
    # Purpose: produce True if self and other are equal Movie objects
    #          and False otherwise
    # Usage:
    # Movie(188, 8.0, "The Princess Bride", 1987)
    #               == Movie(188, 8.0, "The Princess Bride", 1987) => True
    # Movie(188, 8.0, "The Princess Bride", 1987)
    #               == Movie(188, 8.0, "The Princess Bride", 2012) => False

    def __eq__(self, other):
        return type(self)==type(other) and self.rank==other.rank and \
               self.rating==other.rating and self.title==other.title and \
               self.year==other.year
```

```
    # __ne__: Movie Movie -> bool
    # Purpose: produces False if self and other are equal Movie objects
    #          and True otherwise
    # Usage:
    # Movie(188, 8.0, "The Princess Bride", 1987)
                    != Movie(188, 8.0, "The Princess Bride", 1987) => False
    # Movie(188, 8.0, "The Princess Bride", 1987)
                    != Movie(188, 8.0, "The Princess Bride", 2012) => True

    def __ne__(self, other):
        return not (self==other)

## most_common: (listof int[>=1900, <=2012]) -> int[>=1900, <=2012]
## Purpose: Consumes a list of integers which are years. Produces the year
## that most frequently occurs in the list.

def most_common(listof_years):
    year_count_dict = {}
    for a_year in listof_years:
        if year_count_dict.has_hey(a_year) == True:
            year_count_dict[a_year] += 1
        else:
            year_count_dict[a_year] = 1
    return most_common_simplify(year_count_dict)

## most_common_simplify: (dictof int[>=1900,<=2012] int[>0]) > int[>=1900,<=2012]
## Purpose: Consumes a dictionary where the keys are years and the associated
## values are positive integers. The function produces the key that has the
## largest associated value. If two keys have an equal associated value, the
## larger of the two keys is produced (more recent year).

def most_common_simplify(dictof_years):
    keys = dictof_years.keys()
    most_frequent = 0
    how_many = 0
    for a_year in keys:
        val = dictof_years[a_year]
        if val > how_many:
            most_frequent = a_year
            how_many = val
        elif val == how_many:
            most_frequent = max(most_frequent, a_year)
    return most_frequent
```

```
## dict_filter: (dictof str[nonempty] (listof str)[nonempty]) -> (listof Movie)
## Purpose: Consumes a dictionary that contains the movie names (strings) as
## their keys and a list as their values; this list contains, as its first and
## second elements respecitvely, the rank (str) and rating (str) of the movie.
## Produces a list of Movie objects in the order of rank, from smallest to
## largest, where the moview are from the year that most frequently occur.

def dict_filter(dictof_mobies):
    most_pop_yr = str(dict_folder(dictof_movies))
    listof_movies = dictof_movies.keys()
    filtered_list = filter(lambda x: most_pop_yr in x, \
                           listof_movies)
    movie_list = []
    for a_movie in filtered_list:
        year = int(most_pop_yr)
        val = dictof_movies[a_movie]
        rank = int((val[0])[:-1])
        rate = float((val[1])[:-1])
        a_movie_list = (a_movie.split())[:-1]
        a_movie = reduce(lambda x, y: x+ " "+ y, a_movie_list)
        movie = Movie(rank, rate, a_movie, year)
        movie_list.append(movie)
    return movie_list

## dict_folder: (dictof str[nonempty] (listof str)[nonempty]) -> int[>=1900,<=2012]
## Purpose: Consumes a dictionary where the keys are strings that
## represent movie names and the values are nonempty lists that contain
## two strings, one representing the rank and the other representing the
## rating of the movie. Produces the year (an integer) which has the most
## released films out of all the films in the dictionary keys.

def dict_folder(dictionary):
    listof_key = dictionary.keys()
    listof_years = map(lambda x: (x.split())[-1], listof_key)
    listof_years = map(lambda x: int(x[1:5]), listof_years)
    return most_common(listof_years)

## bstyr_dict_maker: (listof (listof str[nonempty])[nonempty])[nonempty]
##                                -> (dictof str[] (listof str)[nonempty])
## Purpose: Consumes a nonempty list of nonempty strings that contain the movie
## information from the file that has been split by ", ". Produces a
## dictionary where the movie names (str) are keys while the associated
## values are nonempty lists that contain two strings, one representing
## the rank and the other representing the rating of the movie.

def bstyr_dict_maker(listof_movies):
    dstyr_dict = {}
    for a_movie in listof_movies:
        bstyr_dict[a_movie[2]] = a_movie[0:2]
    return bstyr_dict
```

```
## movie_list_maker: (listof (listof str[nonempty])[nonempty])[nonempty]
##                           -> (listof (listof str[nonempty])[nonempty])[nonempty]
## Purpose: Consumes a nonempty list of nonempty lists of nonempty strings.
## In each list within the big list, it concatenates all the elements from
## index 2 to index len(x)-1 where x is the list. In this way, it forms a
## string that represents the movie name. Produces a nonempty list of nonempty
## lists of nonempty strings where, each list is of length 3.

def movie_list_maker(listof_list):
    rest_list = map(lambda x: x[2:len(x)-1], listof_list)
    rest_list = map(lambda x: [(reduce(lambda y, z:\
                                       y+" "+z, x)[:-1])], rest_list)
    first_list = map(lambda x: x[0:2], listof_list)
    listof_movies = []
    for first_half, second_half in zip(first_list, rest_list):
        listof_movies.append(first_half + second_half)
    return listof_movies
```

```
## best_year: str[nonempty] -> (listof Movie)[nonempty]
## Purpose: Consumes a nonempty string, and produces a nonempty list of Movie
## objects.
## Effects: Once the function consumes a nonempty string, it reads the file,
## which contains information about movies (as explained on the question sheet).
## The Movie objects that are produced are in order in terms of rank, and are
## those that are released in the year that most frequently occurs in the
## consumed file. If two years occur equally frequently (and most frequently),
## the Movie objects from the most recent years are produced.
## Examples:
## Suppose best_year("ex1.txt") is executed where the following lines are in
## the file (without ##):
## Rank, Rating, Title, Votes
## 1, 9.2, The Shawshank Redemption (1994), 856,696
## 2, 9.2, The Godfather (1972), 624,904
## 3, 9.0, The Godfather: Part II (1974), 399,971
## 4, 8.9, Pulp Fiction (1994), 668,681
## 5, 8.9, The Good, the Bad and the Ugly (1966), 263,660
## The produced list will be:
## [Movie(1, 9.2, 'The Shawshank Redemption', 1994), Movie(4, 8.9, 'Pulp
## Fiction', 1994)]
## Suppose best_year("ex2.txt") is executed where the following lines are
## in the file (without ##):
## 1, 9.2, The Shawshank Redemption (1994), 856,696
## 2, 9.2, The Godfather (2000), 624,904
## 3, 9.0, The Godfather: Part II (2000), 399,971
## 4, 8.9, Pulp Fiction (1994), 668,681
## 5, 8.9, The Good, the Bad and the Ugly (1966), 263,660
## (for the purposes of this question, The Godfather and The Godfather: Part II
## were released in 2000). the following will be produced:
## [Movie(2, 9.2, 'The Godfather', 2000),
##  Movie(3, 9.0, 'The Godfather: Part II', 2000)]

def best_year(filename):
    movie_file = file(filename, 'r')
    movie_list = movie_file.readlines()
    movie_list = map(lambda x: x.split(' '_, movie_list)[1:]
    dictof_movies = bstyr_dict_maker(movie_list_maker(movie_list))
    listof_movies = dict_filter(dictof_movies)
    final_movies = sorted(listof_movies, key= lambda movie: movie.rank)
    movie_file.close()
    return final_movies

## Tests:
check.expect("1Frequent", best_year("q4.txt"), \
             [Movie(1, 9.2, 'The Shawshank Redemption', 1994), \
              Movie(4, 8.9, 'Pulp Fiction', 1994)])

## For the purposes of the test, The Dark Knight was released in 2003, along
## with LOTR: ROTK
check.expect("2Frequent", best_year("q41.txt"), \
             [Movie(8, 8.9, 'The Dark Knight', 2003), \
              Movie(9, 8.8, 'The Lord of the Rings: \
                The Return of the King', 2003)])
```

3 DrRacket Codes

3.1 List of negative standing students

The following code is written using DrRacket. It produces the list of students with negative standing.

```
;;(produces the list of students with negative standings)
;;
;;*************************************************************************
;;
;; A student-record is a list of the form (list id cav passed failed).
;; where
;; id is a positive integer (student id number)
;; cav is an number between 0 and 100, inclusive (student's cumulative
;; average)
;; passed is a non-negative number (number of credits student has passed)
;; failed is a non-negative number (number of credits student has failed)
;;
;; negative-standings; (listof student-record) -> (listof nat)
;; Purpose: produces a list of student ids that have a negative
;; standing by consuming students
;; Examples: (negative-standings (list (list 122 85 9 1)
;;                                     (list 132 55 6 1))) => (list 132)
;; (negative-standings flint (list 132 64 5 3)
;;                           (list 122 59 5 1))) => (list 132 122)

(define (negative-standings students)
    (local
        ;; credits: student-record -> boolean
        ;; Purpose: produces true if the failed credits is not more than one
        ;; half of the passed credits by consuming record.
        [(define (credits record)
            (cond
                [(>= (* 0.5 (third record)) (fourth record)) true]
                [else false]))
         ;; which-standing?: student-record -> Boolean
         ;;Purpose: produces true if the student has a negative standing
         ;:by consuming record
         (define (which-standing? record)
            (cond
                [(and (>= (second record) 80) (credits record)) false]
                [(and
                    (and (>= (second record) 60)
                         (< (second record) 80))
                    (credits record)) false]
                [(< (second record) 60) true]
                [else true]))
```

```
         ;; test-pass: (listof student-record) -> (listof nat)
         ;; Purpose: Produces the student records that have a negative standing by
         ;; consuming records

         (define (test-pass records)
            (filter which-standing? records))

         ;; user-id: student-record => nat
         ;; Purpose: produces the first element of the list by consuming record

         (define (user-id record)
            (first record))]
        (map user-id (test-pass students))))

;; Tests for negative-standings:
(check-expect (negative-standings (list
                                   (list 147 80 5 0)
                                   (list 131 67 10 1))) empty)

(check-expect (negative-standings (list
                                   (list 121 55 9 5)
                                   (list 111 81 5 3)
                                   (list 144 80 5 1))) (list 121 111))
```

3.2 Hangman Game function

The following code is written using DrRacket. It produces a function to play the hangman game.

```
;; (Produces the function to play the hangman game)
;;*********************************************************************************

;; State variables for questions 3 and 4
(define correct-answer void) ; phrase to be guessed
(define current-guess void) ; current state of the player's guess
(define previous-guesses void) ; string containing the letters guessed by the player
(define guesses-left void) ; number of guesses the player has left

;; Constants
(define winning-msg You have correctly guessed the puzzle.")
(define wrong-guess-msg "That is not the correct answer.")
(define good-letter-msg "Good guess.")
(define bad-letter-msg "Letter is not in the puzzle.")
(define previously-chosen-msg "You already tried that letter.")
(define invalid-msg 'Invalid guess.")
(define game-over-msg "No more guesses. Game over.")
(define alphZ 90)
(define alphA 65)

;; Reset/Restart the game
(define (reset-game puzzle level)
    (local
        [(define (level-setter level)
            (cond
                [(symbol=? level 'Beginner) 10]
                [(symbol=? level 'Intermediate) 6]
                [else 3]
                ))
         (define (letter-hider letters)
            (cond
                [(equal? letters #\space) #\space]
                [else #\-]))
         (define (word-hider word)
            (list->string
             (map letter-hider (string->list word)))
            )]
        (begin
            (set! correct-answer puzzle)
            (set! current-guess (word-hider puzzle))
            (set! previous-guesses "")
            (set! guesses-left (level-setter level))
            )))
```

```
;; make-guess: string[nonempty] -> string
;; Consumes a non-empty string called attempt and produces a string, which is
;; a message that comments on the performance and changes the state variables,
;; as per the rules of the game.
;; Effects:
;; When make-guess is evaluated with a given attempt (a string), the variables that
;; may be changed are current-guess, previous-guesses and guesses-left, as per the
;; rules of the game. correct-answer will not be changed unless the game is reset
;; with a different puzzle. As per the rules of the game, for every incorrect answer,
;; guesses-left will decrease by 1 until either the puzzle is solved (when
;; current-guess equals the correct-answer) or guesses-left is 0.
;; Examples:
;; For the following examples, evaluate (reset-game HELLO WORLD Intermediate). Hence,
;; guesses-left is 6. Evaluating (make-guess B) changes the guesses-left to 5,
;; previous-guesses to B, and produces Letter is not in the puzzle. Evaluating
;; (make-guess BOB) changes guesses-left to 4 and produces  That is not the
;; correct answer. previous-guesses is not changed. Evaluating (make-guess L) changes
;; current-guess to --LL- ---L-, does not change guesses-left or previous-guesses and
;; produces Good guess. Evaluating (make-guess T) changes guesses-left to 3,
;; previous-guesses to TLB and produces Letter is not in the puzzle. Evaluating
;; (make-guess B) again does not change any state variables but produces You already
;; tried that letter. Suppose guesses-left is 1; then evaluating (make-guess P)
;; changes guesses-left to 0, previous guesses to PTLB, and produces No more guesses.
;; Game over. Suppose guesses-left is 1; then evaluating (make-guess HELLO WORLD)
;; changes current-guess to correct-answer and produces You have correctly guessed
;; the puzzle.

(define (make-guess attempt)
    (local
        [;; upcase-char?: char-> boolean
         ;; Consumes a character and produces true if it is an upper case alphabetical
         ;; character. Otherwise, produces false.
         (define (upcase-char? char)
            (cond
                [(and
                (<= (char->integer char) alphZ)
                (>= (char->integer char) alphA)) true]
                [else false]))

         ;; str-to-lst-maker: string -> (listof char)
         ;; Consumes a string and produces a list of characters, corresponding to
         ;; each individual letter of the string, in the order as it appears in
         ;; the string
         (define (str-to-lst-maker str)
            (string->list str))

         ;; guess-left-decreaser: (void) -> (void)
         ;; Produces (void).
         (define (guess-left-decreaser)
            (set! guesses-left (sub1 guesses-left)))
```

```
;; msg-gues-lft: (void) -> string
;; Consumes nothing and produces a string, that gives the message associated
;; with however many guesses are left in the game; No more guesses. Game
;; over if zero guesses are left, and Letter is not in the puzzle. otherwise.
(define (msg-gues-lft)
   (cond
       [(equal? guesses-left 1) (begin
                                  (guess-left-decreaser)
                                  game-over msg)]
       [else (begin
               (guess-left-decreaser)
               bad-letter-msg)]))

;; char-replacer: char (listof char) (listof char) -> (listof char)
;; Consumes a character and two lists, and produces a list of characters
;; where the character is inserted in positions in the new list where it
;; occurs in the second list
(define (char-replacer char listx listy)
   (cond
       [(empty? listy) empty]
       [(equal? char (first listy))
        (cons char (char-replacer char (rest listx) (rest listy)))]
       [(equal? (first listx) (first listy))
        (cons (first listx) (char-replacer char (rest listx) (rest listy)))]
       [else (cond #\- (char-replacer char (rest listx) (rest listy)))]))

;; correct-guess: char (listof char) (listof char) -> string
;; Consumes a character, a list of characters representing the current guess,
;; and a list of characters representing the correct answer and produces a
;; string, which comments appropriately
(define (correct-guess phrase guesslist correctlist)
   (begin
       (set! current-guess
           (list->string
            (char-replacer phrase guesslist correctlist)))
       (cond
           [(string=? current-guess correct-answer) winning-msg]
           [else (begin
                   (set! previous-guesses
                       (list->string
                        (cons (first (str-to-lst-maker attempt))
                              (str-to-lst-maker previous-guesses))))
                   good-letter-mg)])))
```

```
;; phrase-correct?: string -> string
;; Consumes a string, checks to see if it is equal to the correct answer
;; and produces the appropriate comment, which is a string
(define (phrase-correct? phrase)
   (cond
       [(string=? phrase correct-answer)
        (begin
           (set! current-guess correct-answer) winning-msg)]
       [else (begin
               (guess-left-decreaser)
               (cond
                   [(< guesses-left 0) (begin
                                         (set! guesses-left 0)
                                         game-over-msg)]
                   [(= guesses-left 1) game-over-msg]
                   [else wrong-guess-msg]))]))]
(cond
    [(= (string-length attempt) 1)
     (cond
        [(not (upcase-char?
                (first (str-to-lst-maker attempt))))
            invalid-msg]
        [else
            (cond
                [(member? (first (str-to-lst-maker attempt))
                          (str-to-lst-maker previous-guesses))
                          previously-chosen-msg]
                [(not
                    (member? (first (str-to-lst-maker attempt))
                                    (str-to-lst-maker correct-answer)))
                 (begin
                    (set! previous-guesses
                        (list->string
                         (cons (first (str-to-1st-maker attempt))
                               (str-to-lst-maker previous-guesses))))
                   (msg-gues-lft))]
                [else (correct-guess
                    (first (str-to-lst-maker attempt))
                    (str-to-lst-maker current-guess)
                    (str-to-1st-maker correct-answer))
                        ]))]
    [else (phrase-correct? attempt)]
    )))
```

```
;; Tests:
(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (and (equal? (make-guess "a") "Invalid guess.")
             (equal? previous-guesses "")
             (= guesses-left 3))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
            (and (equal? (make-guess "B") "Letter is not in the puzzle.")
                 (equal? previous-guesses "B")
                 (= guesses-left 2))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "P")
        (make-guess "B")
        (and (equal? (make-guess "Q") "No more guesses. Game over.")
             (equal? previous-guesses "QBP")
             (= guesses-left 0))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "B")
        (and (equal? (make-guess "B") "You already tried that letter.")
             (equal? previous-guesses "B")
             (= guesses-left 2))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "B")
        (and (equal? (make-guess "E") "Good guess.")
             (equal? previous-guesses "B")
             (equal? current-guess "--EE- E--- --- ---')
             (= guesses-left 2))) true)
```

```
(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "R")
        (make-guess "G")
        (make-guess "A")
        (make-guess "H")
        (make-guess "M")
        (make-guess "N")
        (make-guess "D")
        (make-guess "S")
        (and (equal? (make-guess "E") "You have correctly guessed the puzzle.")
             (equal? previous-guesses "SDNMHAGR")
             (equal? current-guess "GREEN EGGS AND HAM")
             (= guesses-left 3))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (and (equal? (make-guess "GREEN EGGS AND HAM")
                     "You have correctly guessed the puzzle.")
             (equal? previous-guesses "")
             (equal? current-guess "GREEN EGGS AND HAM"))) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "B")
        (make-guess "P")
        (and (equal? (make-guess "BLUE EGGS AND HAM')
                     "No more guesses. Game over.")
             (equal? previous-guesses 'PB"))
             (= guesses-left 0)) true)

(check-expect
    (begin
        (reset-game "GREEN EGGS AND HAM" 'Advanced)
        (make-guess "B")
        (and (equal? (make-guess "BLUE EGGS AND HAM')
                     "That is not the correct answer.")
             (equal? previous-guesses "B"))
             (= guesses-left 1)) true)
```