\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Team Dijkstra  
Crist – Howell – Ghodratnama

t17 – Improvement Project  
Word Games

Software Engineering II  
Spring 2013  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*“Elegance is not a dispensable luxury but a quality that decides between success and failure”   
-* Edsger Dijkstra

Table of Contents

[Overview 4](#_Toc355031533)

[Data Structure Design 4](#_Toc355031534)

[Component Design 5](#_Toc355031535)

[Create Board 5](#_Toc355031536)

[Check Board 6](#_Toc355031539)

[Placement 7](#_Toc355031542)

[Input/Output 10](#_Toc355031557)

[Input 10](#_Toc355031558)

[List of words 10](#_Toc355031559)

[Game Board 10](#_Toc355031560)

[Seed 10](#_Toc355031561)

[Output 11](#_Toc355031562)

[New Game Board 11](#_Toc355031563)

[Probe Estimate 12](#_Toc355031564)

[Source Code 16](#_Toc355031565)

[Create-board.lisp 16](#_Toc355031566)

[Check\_Board.lisp 17](#_Toc355031567)

[Placement.lisp 19](#_Toc355031568)

# Overview

We are modifying Team Van Rossum’s final software product. We will be implementing diagonal word placement and intersecting words in the word search game. In order for us to do this, we will need to work with the base code and replace and update many functions originally written by Van Rossum. This project will require us to redesign the original data structure and include several modification to the main logic provided by Team Van Rossum.

# Data Structure Design

The data structure we will mainly be working with is the game board data structure. This structure will contain a game board, a list of words, and the solutions to the puzzle once they are found.

The format of the game board will be as follows

((#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.))

Where each element in the outer list is a row in the game board. Each element that represents a row will be a list of strings. Initially, each element is originated as a period(.) character. This will allow us to see which spaces are open in the game board when we go to place words. This will also aide in random letter placement when the word placement is complete and random letters are entered.

The next structure that will need to generated, is the structure that contains both the game board, and list of words that will appear in the completed board. This structure will look like:

**(**list '((#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

(#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.))

‘(“word1” “word2” … “wordn”)**)**

This structure contains a list of two elements, the first is the game board structure, the second is a list of strings that represent the words that will appear in the board.

# Component Design

After inspection of Van Rossum’s code, we determined that some of the original modules will have to be completely rewritten to include the new features we want to implement. This is due the narrow linear word placement design that was originally coded. This caused the code base to be heavily dependent on each module and a small change at the core of the computations would not be possible since the slightest changes in code seem to throw off the logic before it gets to the user.

Below are the modules we have worked with in order to implement the new features along with the functionality that we need to implement.

## Create Board

Upon inspection, we were able to determine that the Create Board module would be the initial entry point into creating a word search game. This class contained the entry function (create-board (words game)). This will be the only function that will be left in this module since the overhead functions and other modules use this function to interact with this module. After that, we will replace the module with actual dynamic board sizing that will expand if words do not fit in the default word board size, and take the list of words and place them on the board and fill the board with random letters.

The original module contained several functions that were not necessarily needed to generate a word search board. Most of the functions were used to define their original coordinate structure. This coordinate structure was too confusing and impractical. We replaced the coordinate structure with the standard X,Y coordinate grid for matrices. This allowed us to delete several functions and reduce the module from around 100 lines of code to about 20 lines of code.

The functions that are implemented in this module are:

### (create-board (words game))

This function remains virtually unchanged from the original implementation. This function takes in the two parameters, and calls the appropriate function based on the game variable. When the game variable is a 1, it will call the word search generator. When a 2 is called, it returns nil, since Van Rossum had in their original design, the functionality of implementing crosswords as well. This can only be assumed that this would have called the crossword generator if implemented.

### (wdsrch-brd (words))

This function is the delegator function for the word search generation. This function will call the other various modules in the project and generate a complete game board. First this function will need to generate an empty game board. This is implemented as a standard 6x6 matrix filled with periods.

Once a game board is generated, it is handed to the Placement module where the user’s words are placed into the word search. If a user’s words are too big for the default matrix, this is where the size will by dynamically increased to accommodate larger words.

Once the user’s words are placed into the game board and the final size is determined, we then pass it to the Fill Board module where random letters are placed in the remaining slots. Once this step is complete, the function will return a list where the first item is the game board matrix, and the second item is the solution locations for each word.

## Check Board

The check board module will be responsible for checking that a word will fit in the game board at the selected location in the board and at the selected direction. This is another module that had to be completely rewritten to accommodate our changes in order to return the correct data structure. However, we still ended up with about 40 lines of code compared to their 60 lines of code.

This function will check to see that the solutions will fit into the board at selected locations. If locations for the words can be found, we will return true, if not, we will check to see if another word in the user’s word list can be placed. This process will repeat until a solution can be placed into the board, or we have exhausted the user’s word list. If this occurs, we return this to the calling function and expand the board.

The functions we implemented in this module are:

### (along-axis (check-x check-y next-x next-y direction duration)

This function will traverse the path of the game board and check for spaces that the word can be placed in. This function will take the coordinates and check the next coordinates in order to see if the space is available. If it is, we will continue on the path of the given direction and decrement the duration. This process will continue until the decrement is zero. Once this happens, check-x,y and next-x,y should be equal. Once we reach this state, the word will fit in the specified location on the board and we will return true. Otherwise, if we cannot meet these conditions, the function returns nil and is handled appropriately.

### (check-solution-entry (x y letter solutions))

This function is the entry point and main delegator for this module. This function will check the solutions list to see if a word is able to be placed in the current word search board. This function will check the first word in the list of solutions to see if they can fit into the game board. If they can, we will then call the (along-axis) function to traverse the path of the word to verify the placement of the word is correct.

Based on the X,Y coordinate given, the function will see if the first word in the list is placed at the location. It will check the surrounding spaces in the grid and see if any subsequent letters are found in the word. If the word is not part of the solution at the given X,Y, then we move to the next word in the list until all words are exhausted. At this point, we will return nil since the current space cannot be part of any solution, true otherwise.

## Placement

This is the module where most of our efforts were dedicated. We had originally attempted to reuse the base code and just add our functionality to this module since it was the largest module in the code base. However, due to unforeseen troubles with the data structure format, every time we made a change to this module, it would cause an error once the completed data structure is passed back to the user. This is due to the fact that Placement is at the end of the function calls for the program and one off errors caused the need for this module to be rebuilt from the ground up.

The purpose for this module is to take the user’s list of words and place them in the game board in some random manner. Since true randomness is impossible, Dracula’s rand teachpack was used to have some sense of randomness in the placement of words in the game board. Once the words have been placed into the game board, we will return the completed game board and solutions list. The solutions list will contain the word and the X,Y coordinates of the first and last letter’s location in the matrix.

Each function written and its purpose is listed below:

### (verify-placement)

This function will take in the current game board state, current word, and X,Y coordinate, and a direction. The function will determine whether or not we can place the word at the specified X,Y coordinate with the given direction. To do this, we must check each of the conditions for each letter and locations on the path determined by the direction.

First, we will check to see if we can place the first letter of the word at the given space. This can only be done if the space is empty or the letter matches the letter that is already contained in the space. This process is repeated for each letter in the word. If this process completes successfully we can return true and the word can be placed at this location.

The function will return nil on the following conditions:

* If the words cannot intersect. This is due to collisions in existing words in the board that are already placed and the letters in the word, and in the space are not equal.
* If the word is outside the bounds of the board. This means the location is too close to the edge of the board and we need to find a new location to place this word.
* Finally, if the word reaches beyond the edges of the board and return negative values. This means the game board is too small for the word and will have to be expanded.

### (get-rows-after/before/at)

This is a set of three functions. This functions will return either the set of rows before/after the selected row or the current row if (get-rows-at) is called.

### (get-columns-after/before/at)

This is another set of three functions to where it will return the columns in the game board before/after the selected column or the current column if (get-column-at) is called.

### (replace-characters)

This function will be used to replace the empty characters in the game board with those of the words in the user’s word list after their location have been determined. This function will replace a character at the given row and column. By using the words direction, we can determine the next coordinate to check and place the next character in the word.

We will then repeat this process until the entire word is placed into the game board.

### (place)

This function will take in a word, the game board, direction, and coordinate. This function will extract the coordinate into a row and column number and direction number into a string literal (i.e. a 6 would be converted to “right-up”). Once these conversions are performed, the replace characters function is called with the information for the appropriate direction and the word and game board are passed to the function.

### (get-end-coords)

This function will take the starting coordinates for a word placement and return the ending coordinates based on the direction of the word and the length of the word.

### (is-in-matrix)

This function will take a column and row and the bounds of the game board. From here, we will be able to determine that a row,column pair is inside the bounds of the matrix. If the pair is in the matrix, we will return true, nil otherwise.

### (generate-new-coord)

This function will generate a new random coordinate based on the bounds and a random seed passed into the function.

### (Fit left/right/up/down/right-down/right-up/left-down/left-up)

This is a set of eight functions that will attempt to fit a word into the game board based on the direction of the function called. This function will accept a coordinate, the length of the word, and bounds of the board and see if the word can fit into the board.

This set of functions essentially replaces the Fits module that was originally included with the code.

### (collision)

This function will determine if a collision is occurring in the game board for the current word to be placed. A collision is defined as trying to place a word in the path of another word since this causes overlapping words. If a collision has occurred the function will return that a collision has occurred. If the collision happens to be the same letters, this function will ignore it since these words can be overlapped in the word game.

### (fit-to-board)

This function will determine the correct course of action for adding a word to the board. It will attempt to find a suitable starting location and if it cannot find a suitable location after ten attempts, it will expand the word board to accommodate more spaces for adding words.

This function will have a set of conditions to where the appropriate fitting functions are called. This is based on the direction of the words. Once the location has been confirmed, the staring coordinate will be returned and the placement of the word will commence.

### (append row/column)

These two functions are used by the expand board function to be described next. These functions will append a column/row to the board to allow expansion of the board to allow large words to be appropriately fitted into the game board.

### (expand-board)

This function will append rows and columns to the board to allow larger words to be added to the board. Each expansion will add one row and column to the board. Once this is done, we will try placement again and if the words fit in this expanded board, the board will be completed. If not, we will expand the board again until we have all the words placed into the board.

### (plc-wdsrch)

This function is the main entry point for the module. The name of this function remains the same name as the original code since other modules will make calls to this module by calling this function.

This function acts as a delegation function and call each component of the placement module. Once all the functions have returned without errors, this function will piece together the output data structure and pass this to the calling function.

# Input/Output

Since we are only working with the Placement ACL2 module, we are only going to be working with ACL2 data structure Input and Output. We will go over what will be passed into the module as well as the result that is eventually passed to PHP and displayed to the user.

## Input

In order to execute the Placement.lisp module, the calling function will need to use the “plc-wdsrch (words brd seed)” function in this module. The following are the parameters needed for executing this module.

### List of words

This list will be the list of words you will want to place into the game board. The format of this list will be:

(list “word1” “word2” … “wordn”)

### Game Board

The second part of the input for this module will be the game board. The game board will need to be encapsulated in a list accompanied by its solutions. This format will look like:

(list (list (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)) '())

### Seed

The seed will be the entry point for the random number generator. This number can be chosen by the rand.lisp teachpack by calling (initial-seed) or you may choose an integer of your own.

## Output

The output will be a new game board with the words placed instead of periods (.) in the game board along with the location of the start of each word in the solution list following the game board.

### New Game Board

The completed game board that will be passed to the PHP module will need to look like:

( ((#\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\w #\o #\r #\l #\d #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\l #\. #\. #\. #\. #\. #\t #\. #\.)

( #\. #\. #\. #\. #\l #\. #\. #\. #\. #\s #\. #\. #\.)

( #\. #\. #\. #\. #\e #\. #\. #\. #\e #\. #\. #\. #\.)

( #\. #\. #\. #\. #\h #\. #\. #\t #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)

( #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.))

((hello (5 4)(5 1))

(world (3 1) (7 1))

(test (7 5) (10 2)))

# Probe Estimate

During the initial design phase of this project, we created estimates based on our initial designs. The new objects section of the PSP++ report contain the information of these modules and the projected, detailed estimate for each object.

The estimations provided were calculated using our team's Lines of Code table from the compilation of all the projects from the fall semester.

The table is as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function Type | Tiny 7% | Small 24% | Medium 38% | Large 24% | Huge 7% |
| Non IO Function | 3 | 4 | 6 | 9 | 18 |
| IO Function | 4 | 5 | 9 | 14 | 16 |
| Properties | 3 | 4 | 6 | 8 | 14 |
| Check Expectes | 2 | 3 | 4 | 8 | 14 |

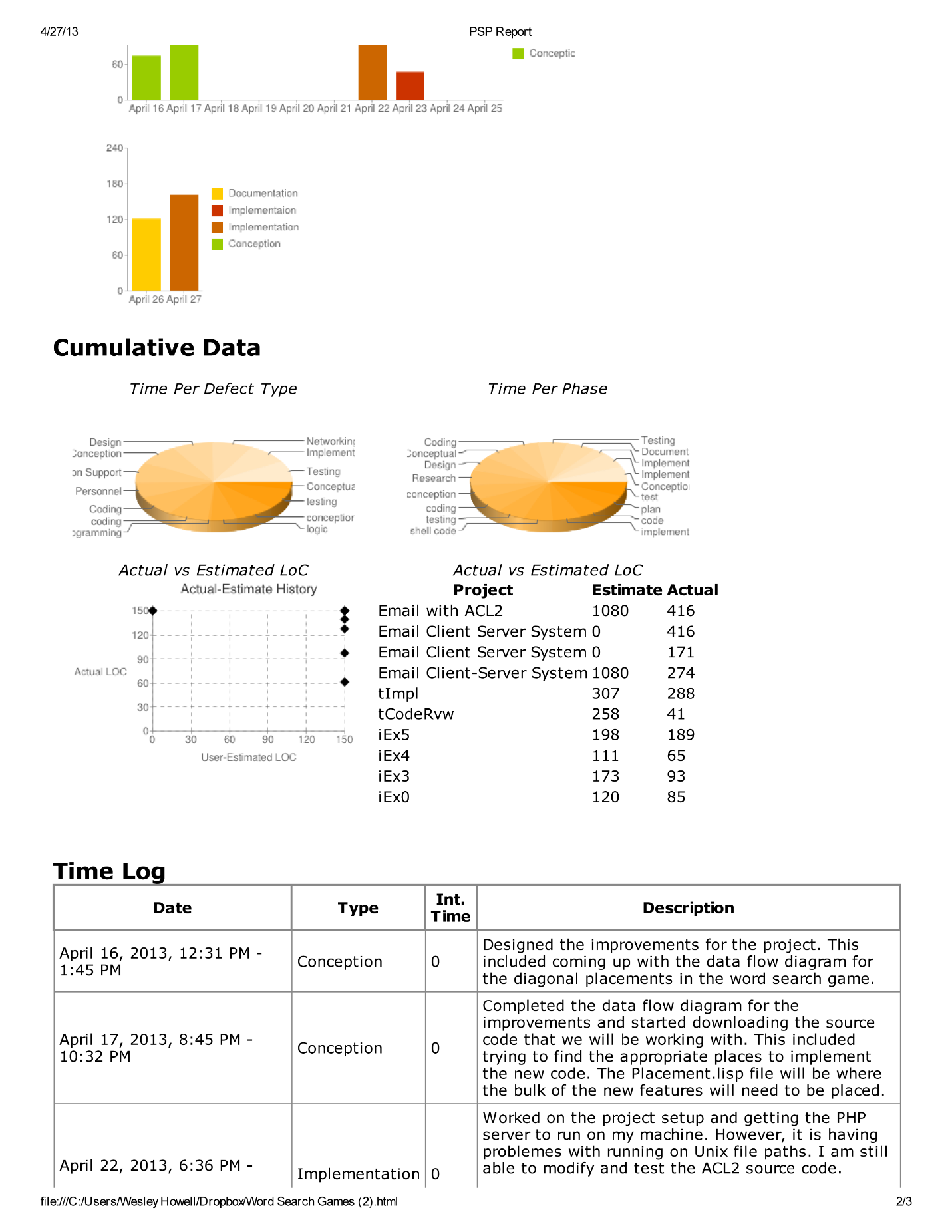
Using this table, we were able to assign an estimated size of each function defined in this project. Each function is listed below with the estimated size and Lines of Code for each particular function.

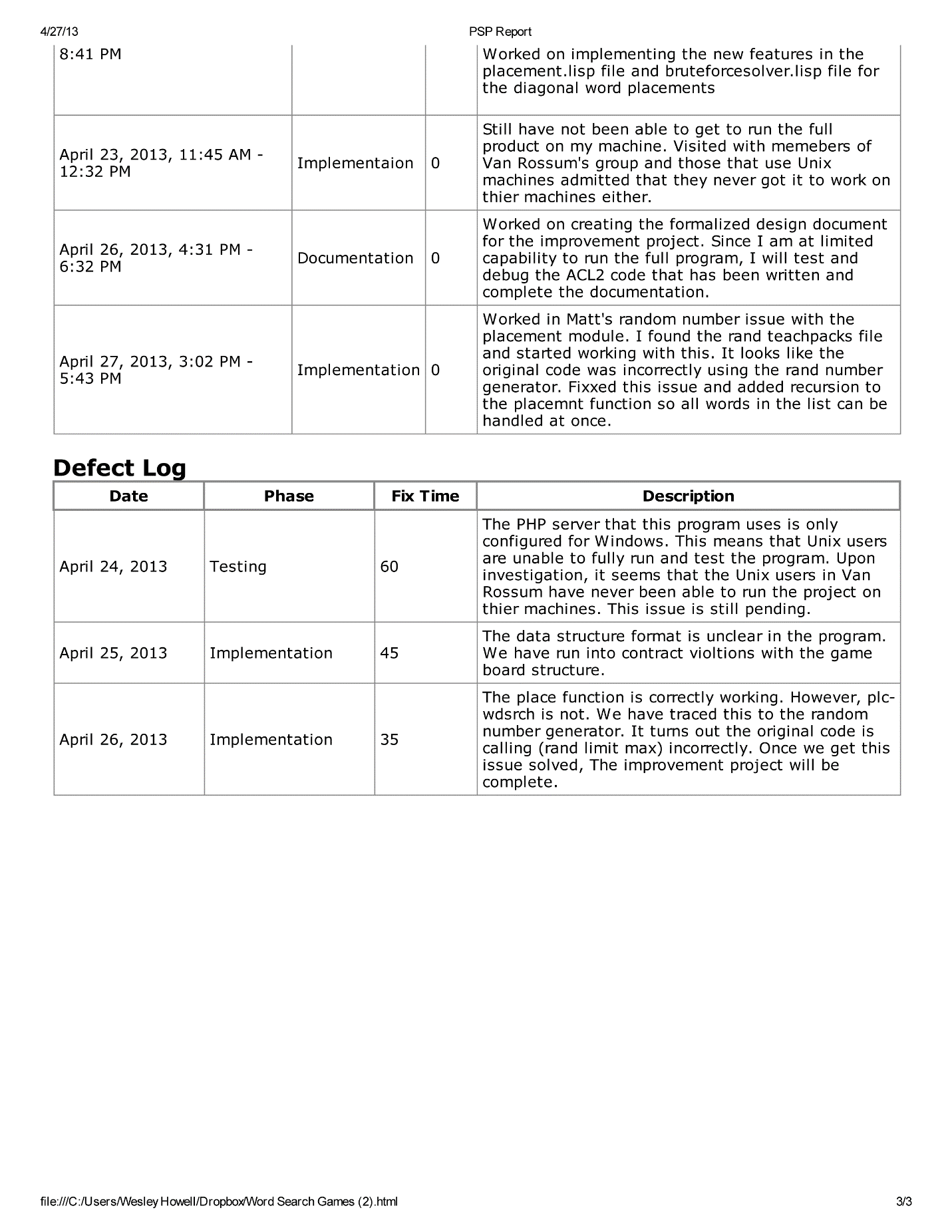
Totaling all these functions, we are able to predict that this project will be around 141 lines of new code.

We also need to note that the aim for this project is 300-500 lines of code. We can accept this 141 LOC estimate because of the margin for error with the PROBE method. Since most of the projects used in the fall semester were small in scale to these projects, we can assume that the PROBE estimate can be off by a **factor up to 3.** Since we are off by a factor of around 2, we can accept this estimate.

The PSP file that contains the objects used for the PROBE estimate is listed below along with the actual Lines of Code added for comparison.







# Source Code

## Create-board.lisp

; Team Team Dijkstra

; create-board.lisp

;

; creates the game board

;

(in-package "ACL2")

(include-book "list-utilities" :dir :teachpacks)

(include-book "placement")

(include-book "fill-board")

;Generate Board for word-search

(defun wdsrch-brd (words)

(let\* ((seed1 1111)

(board-struct (plc-wdsrch words '(((#\. #\. #\. #\. #\. #\.) (#\. #\. #\. #\. #\. #\.) (#\. #\. #\. #\. #\. #\.) (#\. #\. #\. #\. #\. #\.) (#\. #\. #\. #\. #\. #\.) (#\. #\. #\. #\. #\. #\.))) seed1))

(board (car board-struct))

(solutions (cadr board-struct))

(masked (fill-brd board seed1)))

(list masked solutions)))

; Initial call to create appropriate board

(defun create-board (words game)

(cond ((= game 1) (wdsrch-brd words))

((= game 2) nil)))

; Helper to convert board to string

(defun string-brd-helper (row)

(if (endp row) '()

(cons (chrs->str (list (car row)))

(string-brd-helper (cdr row)))))

; Convert from characters to strings

(defun string-brd (brd)

(if (endp brd) '()

(cons (string-brd-helper (car brd))

(string-brd (cdr brd)))))

## Check\_Board.lisp

(in-package "ACL2")

(defun along-axis (check-x check-y next-x next-y direction duration)

(if (and (= check-x next-x) (= check-y next-y))

t

(if (> duration 0)

(cond ((equal direction "left")

(along-axis check-x check-y (- next-x 1) next-y direction (- duration 1)))

((equal direction "right")

(along-axis check-x check-y (+ next-x 1) next-y direction (- duration 1)))

((equal direction "up")

(along-axis check-x check-y next-x (- next-y 1) direction (- duration 1)))

((equal direction "down")

(along-axis check-x check-y next-x (+ next-y 1) direction (- duration 1)))

((equal direction "left-up")

(along-axis check-x check-y (- next-x 1) (- next-y 1) direction (- duration 1)))

((equal direction "left-down")

(along-axis check-x check-y (- next-x 1) (+ next-y 1) direction (- duration 1)))

((equal direction "right-up")

(along-axis check-x check-y (+ next-x 1) (- next-y 1) direction (- duration 1)))

((equal direction "right-down")

(along-axis check-x check-y (+ next-x 1) (+ next-y 1) direction (- duration 1))))

nil))) ; Not located in this solution term

; letter is the string we are checking (make it length of 1)

; this is the method to call

; The letter is insignificant - Matt

(defun check-solution-entry (x y letter solutions)

(if (endp solutions)

nil ; It is not a part of any solution

(let\* ((next-try (car solutions))

(word (coerce (car next-try) 'list))

(start (cadr next-try))

(start-x (car start))

(start-y (cadr start))

(end (caddr next-try))

(end-x (car end))

(end-y (cadr end))

(placement-type

;; left-down, left-up or left

(cond ((> start-x end-x)

(cond ((> start-y end-y) "left-up")

((< start-y end-y) "left-down")

((= start-y end-y) "left")))

;; right-down, right-up or right

((< start-x end-x)

(cond ((> start-y end-y) "right-up")

((< start-y end-y) "right-down")

((= start-y end-y) "right")))

;; up, down or singlet

((= start-x end-x)

(cond ((> start-y end-y) "up")

((< start-y end-y) "down")

((= start-y end-y) "singlet")))))

(found (along-axis x y start-x start-y placement-type (- (length word) 1))))

(if found

t

(check-solution-entry x y letter (cdr solutions))))))

## Placement.lisp

; @Author Team Dijkstra

;

; placement.lisp

;

; places the words within the word search

;

(in-package "ACL2")

(include-book "list-utilities" :dir :teachpacks)

(include-book "rand" :dir :teachpacks)

; (verify-placement word brd col-num row-num direction)

; Verifies the placement of the word. If the word does not meet any of

; the conditions (such as equivalent intercepts or board boundaries)

; then the function returns nil and the board must consider another

; placement coordinate or placement format.

; word - the list of characters that represent the word to be placed on

; the board.

; brd - the matrix of letters that represent the board (list of lists of

; characters).

; col-num - the column number for placement.

; row-num - the row number for placement.

; direction - the direction in which the word is to be placed.

(defun verify-placement (word brd col-num row-num direction)

(if (and (natp col-num)

(natp row-num))

(if (and (> (length brd) row-num)

(> (length (car brd)) col-num)

(or (< 0 col-num) (equal col-num 0))

(or (< 0 row-num) (equal row-num 0)))

(if (endp word)

t ; There are no more letters to verify and the meet the conditions

(let\* ((row (nth row-num brd))

(column (nth col-num row)))

(if (or (equal column (car word))

(equal column #\.))

(cond ((equal direction "right-down")

(verify-placement (cdr word) brd (+ col-num 1) (+ row-num 1) direction))

((equal direction "right-up")

(verify-placement (cdr word) brd (+ col-num 1) (- row-num 1) direction))

((equal direction "left-down")

(verify-placement (cdr word) brd (- col-num 1) (+ row-num 1) direction))

((equal direction "left-up")

(verify-placement (cdr word) brd (- col-num 1) (- row-num 1) direction))

((equal direction "down")

(verify-placement (cdr word) brd col-num (+ row-num 1) direction))

((equal direction "up")

(verify-placement (cdr word) brd col-num (- row-num 1) direction))

((equal direction "left")

(verify-placement (cdr word) brd (- col-num 1) row-num direction))

((equal direction "right")

(verify-placement (cdr word) brd (+ col-num 1) row-num direction)))

nil))) ; Word cannot interst (collision issue)

nil) ; Word is outside of the bounds of the board

nil)) ; Row and column do not exist - negative values

; (get-rows-after brd row-num)

; Acquires all rows after a given row-num.

; brd - the board matrix (list of lists) that contain the row information.

; row-num - the number of the row after which you wish to acquire rows.

(defun get-rows-after (brd row-num)

(if (equal row-num 0)

(cdr brd)

(get-rows-after (cdr brd) (- row-num 1))))

; (get-rows-before brd row-num)

; Acquires all rows before a given row-num.

; brd - the board matrix (list of lists) that contain the row information.

; row-num - the number of the row before which you wish to acquire rows.

(defun get-rows-before (brd row-num)

(if (equal row-num 0)

'()

(append (list (car brd)) (get-rows-before (cdr brd) (- row-num 1)))))

; (get-row-at brd row-num)

; Acquires the row at a given row-num.

; brd - the board matrix (list of lists) that contain the row information.

; row-num - the row in which you wish to acquire from the matrix.

; \*\* It is important to note that this returns the row tuple and not a

; list of lists like the previous boundary acquisition functions.

(defun get-row-at (brd row-num)

(if (equal row-num 0)

(car brd)

(get-row-at (cdr brd) (- row-num 1))))

; (get-cols-after row col-num)

; Acquires the columns that occur after a given col-num.

; row - the list of columns in a row.

; col-num - the column after which you wish to acquire the columns.

(defun get-cols-after (row col-num)

(if (equal col-num 0)

(cdr row)

(get-cols-after (cdr row) (- col-num 1))))

; (get-cols-before row col-num)

; Acquires the columns that occur before a given col-num.

; row - the list of columns in a row.

; col-num - he column before which you wish to acquire the columns.

(defun get-cols-before (row col-num)

(if (equal col-num 0)

nil

(cons (car row) (get-cols-before (cdr row) (- col-num 1)))))

; (replace-characters word brd col-num row-num direction)

; Replaces a character at the given row and column. The direction determines

; the next replacement coordinate until all the letters in the word have been

; placed onto the board.

; word - the list of characters that will be placed on the board.

; brd - the list of lists of characters that currently make up the board.

; col-num - the column index that is to be replaced.

; row-num - the row index that is to be replaced.

; direction - the direction that the characters will be replaced (next index).

(defun replace-characters (word brd col-num row-num direction)

(if (endp word)

brd ; We have no more letters to place

(let\* ((front (get-rows-before brd row-num))

(back (get-rows-after brd row-num))

(change (get-row-at brd row-num))

(fcol (get-cols-before change col-num))

(bcol (get-cols-after change col-num))

(nrow (append fcol (cons (car word) bcol)))

(nbrd (append front (cons nrow back))))

(cond ((equal direction "right") (replace-characters (cdr word) nbrd (+ col-num 1) row-num direction))

((equal direction "left") (replace-characters (cdr word) nbrd (- col-num 1) row-num direction))

((equal direction "up") (replace-characters (cdr word) nbrd col-num (- row-num 1) direction))

((equal direction "down") (replace-characters (cdr word) nbrd col-num (+ row-num 1) direction))

((equal direction "right-down") (replace-characters (cdr word) nbrd (+ col-num 1) (+ row-num 1) direction))

((equal direction "right-up") (replace-characters (cdr word) nbrd (+ col-num 1) (- row-num 1) direction))

((equal direction "left-down") (replace-characters (cdr word) nbrd (- col-num 1) (+ row-num 1) direction))

((equal direction "left-up") (replace-characters (cdr word) nbrd (- col-num 1) (- row-num 1) direction))))))

; (place brd word type coord)

; Place word on the board at the given coordinate in the direction

; specified. We first do a dry run with verify-placement and if

; that function returns true, we are able to place our word onto the

; board without conflict. If we are not able to place it on the board,

; we return the original board. \*\* This is intended to be replaced by

; randomly selecting another place for the word on the board.

; brd - the board in which to place the word.

; word - the word that will be placed into the board.

; type - the orientation of placement for the word.

; coord - the coordinate (col row) to place the word.

(defun place (brd word type coord)

(let\* ((column (car coord))

(row (cadr coord)))

; Right placement

(cond ((= type 0) (replace-characters word brd column row "right"))

; Left placement

((= type 1) (replace-characters word brd column row "left"))

; Down placement

((= type 2) (replace-characters word brd column row "down"))

; Up placement

((= type 3) (replace-characters word brd column row "up"))

; Right-Down placement

((= type 4) (replace-characters word brd column row "right-down"))

; Left-Down placement

((= type 5) (replace-characters word brd column row "left-down"))

; Right-Up placement

((= type 6) (replace-characters word brd column row "right-up"))

; Left-Up placement

((= type 7) (replace-characters word brd column row "left-up")))))

; (get-end-coords start-coord word-length orientation)

; Acquires the endpoint coordinate of the word based on its placement on the board.

; start-coord - the x,y tuple (col, row) where the word is started

; word-length - the length of the word that was placed

; orientation - the integer representation for orientation - see below for int values

(defun get-end-coords (start-coord word-length orientation)

; Right placement - shift x, keep y

(cond ((= orientation 0) (list (+ (car start-coord) (- word-length 1)) (cadr start-coord)))

; Left placement - shift x, keep y

((= orientation 1) (list (- (car start-coord) (- word-length 1)) (cadr start-coord)))

; Downward placement - shift y, keep x

((= orientation 2) (list (car start-coord) (+ (cadr start-coord) (- word-length 1))))

; Upward placement - shift y, keep x

((= orientation 3) (list (car start-coord) (- (cadr start-coord) (- word-length 1))))

; Right downward placement

((= orientation 4) (list (+ (car start-coord) (- word-length 1)) (+ (cadr start-coord) (- word-length 1))))

; Left downward placement

((= orientation 5) (list (- (car start-coord) (- word-length 1)) (+ (cadr start-coord) (- word-length 1))))

; Right upward placement

((= orientation 6) (list (+ (car start-coord) (- word-length 1)) (- (cadr start-coord) (- word-length 1))))

; Left upward placement

((= orientation 7) (list (- (car start-coord) (- word-length 1)) (- (cadr start-coord) (- word-length 1))))))

; (is-in-matrix col row bounds)

; This function is used to determine if the point is within the bounds of

; the matrix.

; col - the x coordinate.

; row - the y coordinate.

; bounds - the length of the list of columns of the matrix which should

; be equivalent to the length of the rows in the matrix.

(defun is-in-matrix (col row bounds)

(if (and (or (> col 0) (= col 0))

(< col bounds)

(or (> row 0) (= row 0))

(< row bounds))

t

nil))

; (generate-new-coord bounds seed)

; Generates a new coordinate based on the bounds and the seed that is

; fed to the function.

; bounds - the integer values that determines the limit of the row and

; column. Since the matrix is a perfect square, this is an int

; value to represent both axis'.

; seed - the seed that will be used to determine the next random value

; for placement.

(defun generate-new-coord (bounds seed)

(let\* ((seed1 (next-seed seed))

(col (rand bounds seed1))

(seed2 (next-seed seed1))

(row (rand bounds seed2)))

(list col row)))

; (fit-left coords length bounds)

; Attempts to make this word fit with a left orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-left (coords length bounds)

(let\* ((col (car coords))

(row (cadr coords)))

(if (is-in-matrix col row bounds)

; end-point is equivalent to the overflow in this case

(let\* ((end-point (- col (- length 1))))

(if (< end-point 0)

(let\* ((new-col (+ (abs end-point) col)))

(list new-col row))

(list col row)))

; Use the col as a seed value

(fit-left (generate-new-coord bounds (abs col)) length bounds))))

; (fit-right coords length bounds)

; Attempts to make this word fit with a right orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-right (coords length bounds)

(let\* ((col (car coords))

(row (cadr coords)))

(if (is-in-matrix col row bounds)

(let\* ((end-point (+ col (- length 1))))

(if (> end-point (- bounds 1))

(let\* ((overflow (- (abs end-point) (- length 1)))

(new-col (- col (- overflow 1))))

(list new-col row))

(list col row)))

; Use the col as a seed value

(fit-right (generate-new-coord bounds (abs col)) length bounds))))

; (fit-up coords length bounds)

; Attempts to make this word fit with a up orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-up (coords length bounds)

(let\* ((col (car coords))

(row (cadr coords)))

(if (is-in-matrix col row bounds)

; end-point is equivalent to the overflow in this case

(let\* ((end-point (- row (- length 1))))

(if (< end-point 0)

(let\* ((new-row (+ (abs end-point) row)))

(list col new-row))

(list col row)))

; Use the row as a seed value

(fit-left (generate-new-coord bounds (abs row)) length bounds))))

; (fit-down coords length bounds)

; Attempts to make this word fit with a right orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-down (coords length bounds)

(let\* ((col (car coords))

(row (cadr coords)))

(if (is-in-matrix col row bounds)

(let\* ((end-point (+ row (- length 1))))

(if (> end-point (- bounds 1))

(let\* ((overflow (- (abs end-point) (- length 1)))

(new-row (- row (- overflow 1))))

(list col new-row))

(list col row)))

; Use the row as a seed value

(fit-right (generate-new-coord bounds (abs row)) length bounds))))

; (fit-left-down coords length bounds)

; Attempts to make this word fit with a left down diagonal orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-left-down (coords length bounds)

(fit-down (fit-left coords length bounds) length bounds))

; (fit-right-down coords length bounds)

; Attempts to make this word fit with a right down diagonal orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-right-down (coords length bounds)

(fit-down (fit-right coords length bounds) length bounds))

; (fit-right-up coords length bounds)

; Attempts to make this word fit with a right up diagonal orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-right-up (coords length bounds)

(fit-up (fit-right coords length bounds) length bounds))

; (fit-left-up coords length bounds)

; Attempts to make this word fit with a left up diagonal orientation.

; coords - the tuple (col row) of the placement of a word.

; length - the length of the word that is to be placed.

; bounds - the bound value for the board. \*\* Board is a square so bound

; is a single int that represents the x and y bounds.

(defun fit-left-up (coords length bounds)

(fit-up (fit-left coords length bounds) length bounds))

; (collision graph start-point end-point)

; Detects if there is a collision on the graph with the current word being

; placed into the matrix.

; \*\* If the collision is of like characters, the collision is ignored since

; this would allow for word intersection.

; graph - the matrix that contains all the characters currently placed on

; the matrix.

; start-point - the starting point of a word being checked against the matrix.

; end-point - the ending point of a word being checked against the matrix.

; word - the word that is being placed into the matrix.

(defun collision (graph start-point end-point word)

(let\* ((x1 (car start-point))

(x2 (car end-point))

(y1 (cadr start-point))

(y2 (cadr end-point))

(row (nth y1 graph))

(col (nth x1 row)))

(if (or (equal col #\.) (equal col (car word)))

(cond ((= x1 x2)

(cond ((= y1 y2) nil)

; Right position collision detection

((< y1 y2) (collision graph (list x1 (+ y1 1)) end-point (cdr word)))

; Left position collision detection

((> y1 y2) (collision graph (list x1 (- y1 1)) end-point (cdr word)))))

((< x1 x2)

; Down position collision detection

(cond ((= y1 y2) (collision graph (list (+ x1 1) y1) end-point (cdr word)))

; Down-Right position collision detection

((< y1 y2) (collision graph (list (+ x1 1) (+ y1 1)) end-point (cdr word)))

; Down-Left position collision detection

((> y1 y2) (collision graph (list (+ x1 1) (- y1 1)) end-point (cdr word)))))

((> x1 x2)

; Up position collision detection

(cond ((= y1 y2) (collision graph (list (- x1 1) y1) end-point (cdr word)))

; Up-Right collision detection

((< y1 y2) (collision graph (list (- x1 1) (+ y1 1)) end-point (cdr word)))

; Up-Left collision detection

((> y1 y2) (collision graph (list (- x1 1) (- y1 1)) end-point (cdr word))))))

t)))

; (fit-to-board word matrix orientation seed attempts)

; This function will determine the correct course of action for adding

; a word to the board. It will attempt to find a suitable start location

; and if it cannot find a suitable location after 10 tries, it will

; expand the playing board to accomidate more play tiles for adding

; the word.

; word - the word that will be added to the matrix.

; matrix - the current playing board that will have the word added.

; orientation - the placement orientation of the word.

; seed - the next seed to generate a random number.

; attempts - the incrementer to check attempts of adding word to board.

(defun fit-to-board (word matrix orientation seed attempts)

(if (> attempts 10)

nil

(let\* ((bounds (length matrix))

(length (length word)))

(cond ((equal orientation "right")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-right coord length bounds))

(end-coord (list (+ (car start-coord) (- length 1)) (cadr start-coord))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "left")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-left coord length bounds))

(end-coord (list (- (car start-coord) (- length 1)) (cadr start-coord))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "up")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-up coord length bounds))

(end-coord (list (car start-coord) (- (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "down")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-down coord length bounds))

(end-coord (list (car start-coord) (+ (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "right-down")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-right-down coord length bounds))

(end-coord (list (+ (car start-coord) (- length 1)) (+ (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "right-up")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-right-up coord length bounds))

(end-coord (list (+ (car start-coord) (- length 1)) (- (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "left-down")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-left-down coord length bounds))

(end-coord (list (- (car start-coord) (- length 1)) (+ (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))

((equal orientation "left-up")

(let\* ((seed1 (next-seed seed))

(row (rand bounds seed1))

(seed2 (next-seed seed1))

(col (rand bounds seed2))

(coord (list col row))

(start-coord (fit-left-up coord length bounds))

(end-coord (list (- (car start-coord) (- length 1)) (- (cadr start-coord) (- length 1)))))

(if (collision matrix start-coord end-coord word)

(fit-to-board word matrix orientation seed2 (+ attempts 1))

start-coord)))))))

; (append-columns board)

; This will append an extra column to the end of each row.

; board the current playing surface to be expanded.

(defun append-columns (board)

(if (endp board)

'()

(append (list (append (car board) '(#\.))) (append-columns (cdr board)))))

; (append-row board)

; This will append a row to the board.

; board - the current playing surface to be expanded.

(defun append-row (board)

(append board (list (make-list (length (car board)) :initial-element #\.))))

; (expand-board board)

; This function will add an additional column and row to the playing matrix

; causing the playing surface to expand to accept more words to add to the

; crossword puzzle.

; board - the current playing surface to be expanded.

(defun expand-board (board)

(let\* ((solutions (cadr board))

(new-board (append-columns (append-row (car board)))))

(list new-board solutions)))

; (add-solution solutions solution)

; Adds a solution to the current list of solutions.

; solutions - the list of solutions currently on the board.

; solution - the solution that will be added to the board.

(defun add-solution (solutions solution)

(if solutions

(append solutions (list solution))

(list solution)))

; (plc-wdsrch words brd seed)

; Entry point for the placement module.

; words - a list of words as strings '("word1" "word2" ... "wordn")

; brd - a game board with solutions (list (list (list #\. #\. #\.)

; (list #\. #\. #\.)

; (list #\. #\. #\.))

; '())

; seed - an integer for the random numbers

(defun plc-wdsrch (words brd seed)

(if (endp words)

brd

(let\* ((word (str->chrs (car words)))

(letter-board (car brd))

(solutions (cadr brd))

(seed1 (next-seed seed))

(type (rand 8 seed1))

(seed2 (next-seed seed1))

(orientation (cond ((= type 0) "right")

((= type 1) "left")

((= type 2) "down")

((= type 3) "up")

((= type 4) "right-down")

((= type 5) "left-down")

((= type 6) "right-up")

((= type 7) "left-up")))

(start-coord (fit-to-board word letter-board orientation seed2 0)))

(if start-coord

(let\* ((new-board (place letter-board word type start-coord))

(new-word-solution (list (car words) start-coord (get-end-coords start-coord (length word) type)))

(new-solutions (add-solution solutions new-word-solution))

(new-brd (list new-board new-solutions)))

(plc-wdsrch (cdr words) new-brd (next-seed seed2)))

; We don't have any place to put this word! Expand the matrix.

(plc-wdsrch words (expand-board brd) (next-seed seed2))))))