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

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](#_Toc354853407)

[Data Structure Design 4](#_Toc354853408)

[Component Design 5](#_Toc354853409)

[Placement 5](#_Toc354853410)

[Input/Output 9](#_Toc354853425)

[Input 9](#_Toc354853426)

[List of words 9](#_Toc354853427)

[Game Board 9](#_Toc354853428)

[Seed 9](#_Toc354853429)

[Output 10](#_Toc354853430)

[New Game Board 10](#_Toc354853431)

[Probe Estimate 11](#_Toc354853432)

[Source Code 15](#_Toc354853433)

# Overview

We are modifying Team Van Rossum’s final software product. We will be implementing diagonal word placement 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 is due to the original data structure not being able to have diagonal and intersecting words being placed in the game.

# 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

Upon inspection of the product given by Van Rossum, we are able to determine that our changes will need to occur in two modules, Placement and Brute Force Solver.

Our focus will be in Placement, since we want to be able to generate the games. Once this is finished we will then proceed to the Brute Force Solver. This will allow us to both generate and solve diagonal word games.

## Placement

The placement module is where most of the logic associated with created diagonal word placements will need to occur. In order for us to have successful implementation, we have had to modify several of Van Rossum’s base code. This is due to unclear documentation that was delivered to our team. The new product will need to include the following requirements:

### (verify-placement (word board column-number row-number direction)

This function will need to perform a verification on the letter placement of the supplied word. The function will check to board to see if the supplied column and row positions will be able to contain the word basing the supplied column and row cell is the starting cell of the word. If the board is unable to place the word, then another cell will need to be entered to see if the word can be placed. The parameters needed for this function will be:

* Word – The word that is currently being placed
* Board – The game board structure that we want to place the word into.
* Column-number – The number of the column we want to place the word into
* Row-number – The number of the row we want to place the word into. Combined, the row nad column number create a coordinate which will be the location of the first letter of word.
* Direction – The direction the word is to be placed. (I.E. up, down, diagonal-up-left, diagonal-down-right, …)

### (get-rows-after board row-number)

This function returns the rows in the game board after the current board that is selected

* Board – The game board
* Row-Number – The row that you want to get the rows after.

### (get-rows-before board row-number)

This function will return the rows in the game board that occur before the selected row-number

* Board – The game board
* Row-Number – The row that you want to get the rows before.

### (get-row-at board row-number)

This function will return the selected row in the game board.

* Board – The game board
* Row-Number – The row that you want to get the rows after.

### (get-cols-after columns column-number)

This function will return the columns that occur after the column selected

* Columns – The columns in the game board
* Column-Number – The column you want to retrieve the columns after.

### (get-cols-before columns column-number)

This function will return the columns that occur before the column selected

* Columns – The columns in the game board
* Column-Number – The column you want to retrieve the columns before.

### (replace-characters word board col-num row-num direction)

This function will replace the characters in the game board with the current word at the specified cell. This function will also replace the characters in the game board based on the selected direction of the word.

* Word – The word to add to the game board
* Board – the current game board you are working with
* Col-num – the Y coordinate of the placement of the first letter in the word
* Row-num – the X coordinate of the placement of the first letter in the word

### (plc-rd board word coordinate)

This function will place the characters in the word on the game board in the Right, Downward diagonal on the game board.

* Board – the current game board you are working with
* Word – the word to place on the game board
* Coordinate – the location of the first letter of the word.

### (plc-ld board word coordinate)

This function will place the characters in the word on the game board in the Left, Downward diagonal on the game board.

* Board – the current game board you are working with
* Word – the word to place on the game board
* Coordinate – the location of the first letter of the word.

### (plc-ru board word coordinate)

This function will place the characters in the word on the game board in the Right, Upward diagonal on the game board.

* Board – the current game board you are working with
* Word – the word to place on the game board
* Coordinate – the location of the first letter of the word.

### (plc-lu board word coordinate)

This function will place the characters in the word on the game board in the Left, Upward diagonal on the game board.

* Board – the current game board you are working with
* Word – the word to place on the game board
* Coordinate – the location of the first letter of the word.

### (place board word type coord) – Modified

This was an original function in Van Rossum’s code. This function will check the conditions based on the “type” variable. This variable is a condition that determines the direction of the placement and then cals the respective placement command. The additions we made to this function include:

* Type = 4 is a Right Downward diagonal placement
* Type = 5 is a Left Downward diagonal placement
* Type = 6 is a Right Upward diagonal placement
* Type = 7 is a Left Upward diagonal placement

With these expanded variables, once a condition is met, the program will call one of the four above functions to place a word into the game board.

### (get-end-coords start-coord word-length orientation) – Modified

This is another function that had to be modified for successful implementation of diagonal placements. Based on the “orientation” variable, we can determine what the word’s placement in the game board should be. Using the same scale as “type” in the Place function, we can find the ending coordinate of where the word’s placement in the game board should be.

Once we have obtained the function, we can generate a coordinate pair that marks the start and ending location of the word. This will eventually be included with the results of the function and allow us to see where the word has been placed on the game board.

### (plc-wdsrch words brd seed) – Modified

This function will be the main entry point for the placement module. This function will take care of making sure that each word is placed into the game board and is done so successfully. The result of this function is what is eventually passed the original PHP module for displaying the game board to the user.

This module will require the use of random numbers. We will use the random numbers teachpack to generate pseudo random numbers to be used in this application.

This function will need the following input in order to complete successfully:

* Words – A list of words to be placed in the game board
* Brd – The game board structure with solutions list
* Seed – the seed for the pseudo random number generator

This function will run the recursive part of this function by taking the list of words supplied and place words until the list is empty. Once the list of words is empty, the function will return the game board with placement locations attached in the output.

# 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 |

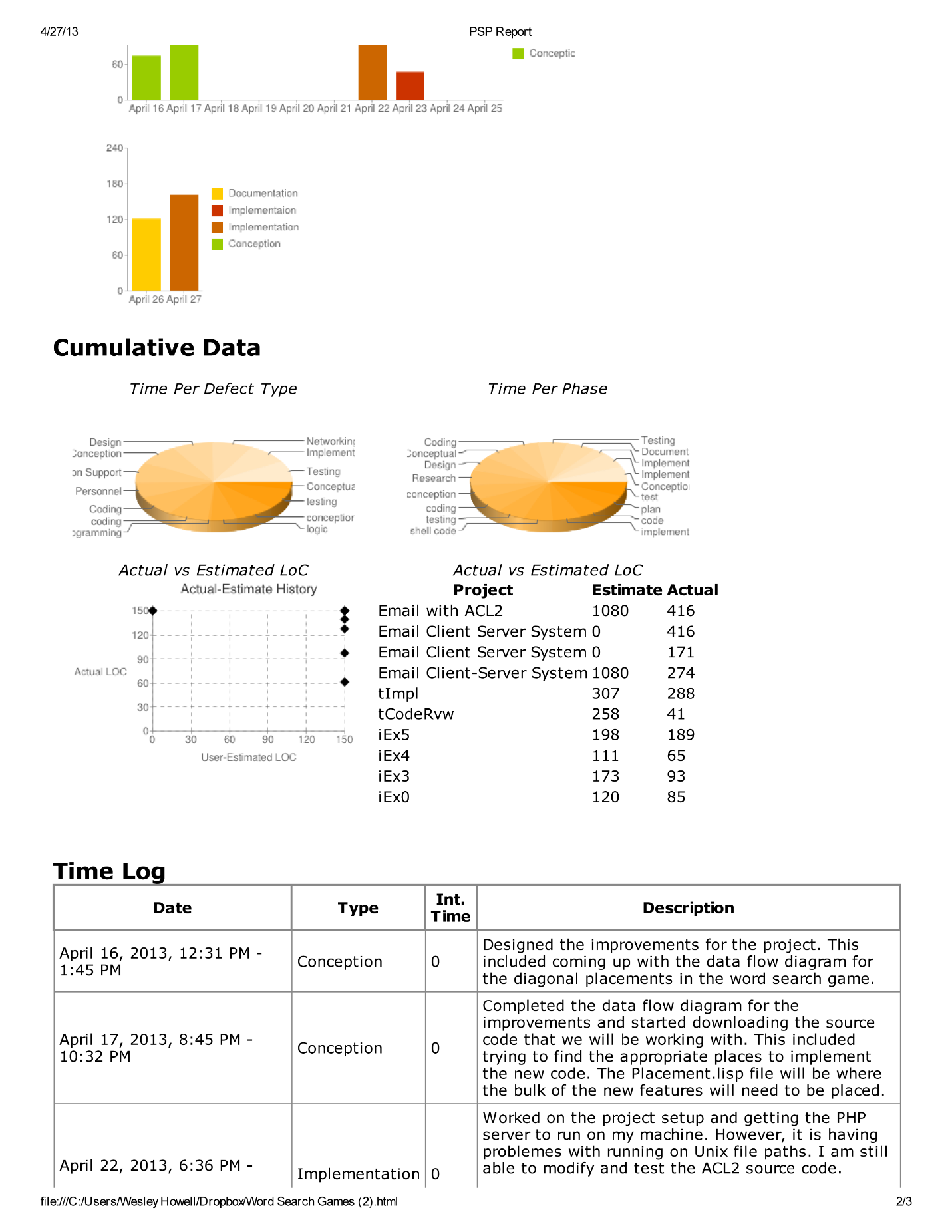
Using this table, we were able to assign an estimated size of each function defined in this project. Each function is listed below with the estimated size and Lines of Code for each particular function.

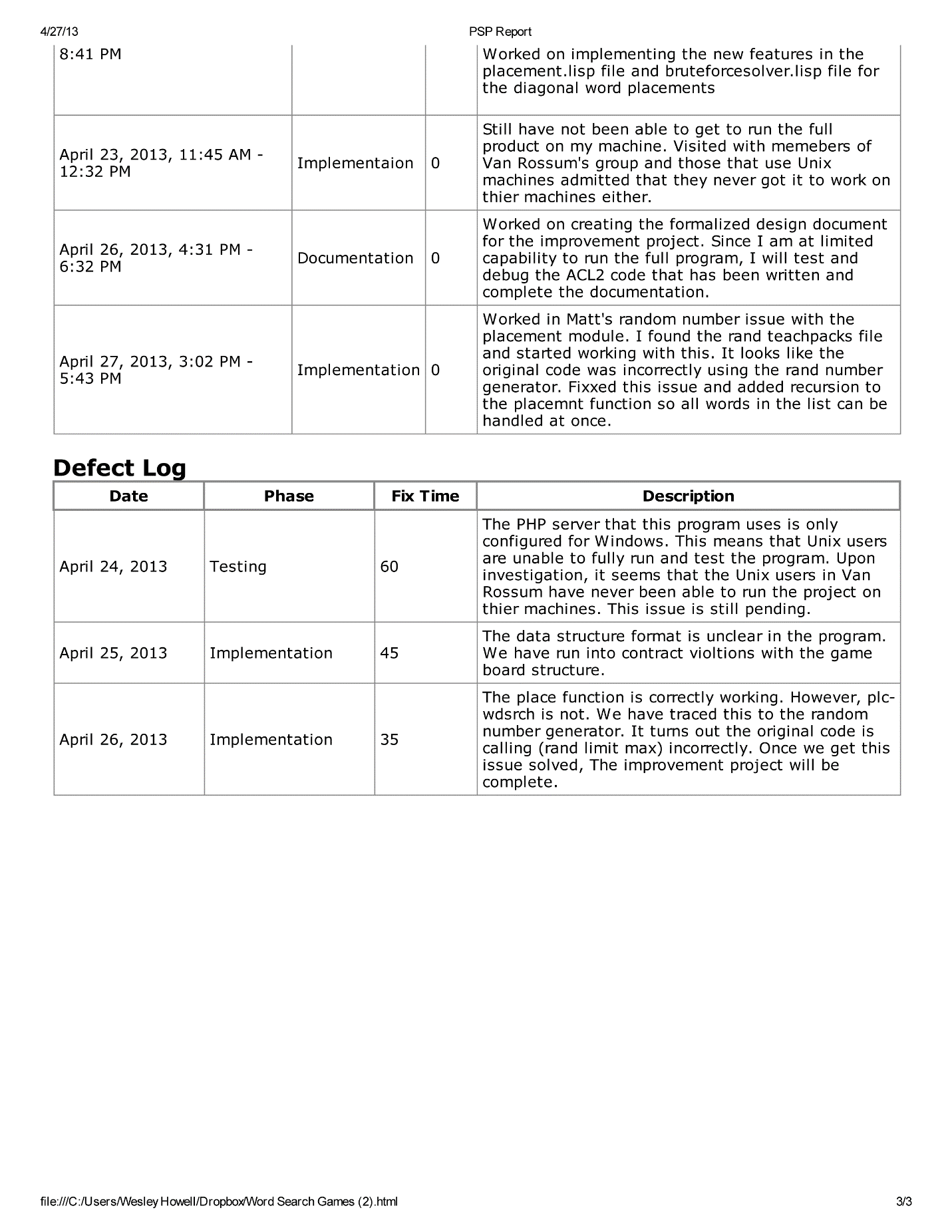
Totaling all these functions, we are able to predict that this project will be around 141 lines of new code.

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

; @Author Van Rossum

;

; placement.lisp

;

; places the words within the word search

;

(in-package "ACL2")

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

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

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

(include-book "fits")

;----------------------------------------------------------------

;---------X,Y placement-----------------------------------------

;---------------------------------------------------------------

;Get specified row of the board

(defun get-row (brd n)

(nth n brd))

; Returns the column for replacement

(defun get-column (brd col)

(if (endp brd) '()

(cons (nth col (car brd))

(get-column (cdr brd) col))))

;Updates the board by putting the modified row

; into its right place making a new board

(defun update-row (brd brd-length row row-num n)

(if (= n brd-length) nil ;finish

(if (= n row-num)

(cons row

(update-row

(cdr brd) brd-length row row-num (+ 1 n)))

(cons (car brd)

(update-row (cdr brd) brd-length

row

row-num

(+ 1 n)))) ))

; Helper method for placing at spec indx

(defun plc-indx-helper (n y row char)

(if (endp row) '()

(if (= n y )

(cons char (plc-indx-helper (+ 1 n) y (cdr row) char))

(cons (car row) (plc-indx-helper (+ 1 n) y (cdr row) char)))))

; Places character at specified index

(defun plc-indx (brd x y char)

(let\* ((row (get-row brd x))

(new-row (plc-indx-helper 0 y row char)))

(update-row brd (len brd) new-row x 0)))

; Replaces column

(defun col-rep (brd chrs y col)

(if (endp chrs) brd

(let\* ((new-brd (plc-indx brd y col (car chrs))))

(col-rep new-brd (cdr chrs) (+ 1 y) col))))

;Replace row

(defun row-rep (brd chrs y col)

(if (endp chrs) brd

(let\* ((new-brd (plc-indx brd y col (car chrs))))

(row-rep new-brd (cdr chrs) y (+ 1 col) ))))

;----------------------------------------------End Board Modifications

; Picks a random starting point for horizontal placement

(defun rand-start-horiz (start range word row-num seed)

(let\* ((new-start (+ start (rand range seed)))

(let\* ((new-start (+ start (rand range seed)))

(new-end (+ new-start (len word))))

(list (list row-num new-start) (list row-num new-end)) ))

; Picks a random starting point for vertical placement

(defun rand-start-vert (start range word row-num seed)

(let\* ((new-start (+ start (rand (- range 1) seed)))

(new-end (+ new-start (len word))))

(list (list new-start row-num) (list new-end row-num))))

;word len = 5

; start 0

; end 10

; diff 10

; new-start = rand( diff - lenword start anywhere between 0, 5

(defun rand-start (coords word seed type)

(if (< type 2)

(let\* ((start (cadar coords))

(end (cadadr coords))

(row-num (caar coords))

(diff (- end start))

(range (- diff (len word))))

(if (<= range 2)

coords

(rand-start-horiz start range word row-num (+ 19 seed)))) ;ugly function

(let\* ((start (caar coords))

(row-num (cadar coords))

(end (caadr coords))

(diff (- end start))

(range (- diff (len word))))

(if (<= range 2)

coords

(rand-start-vert start range word row-num (+ 12 seed))))))

; Randomly picks a coord from coords list for placement

(defun rand-coord (seed coords)

(nth (rand (len coords) seed) coords))

; Function just gathers all needed info

; returns coordinate for where to place

(defun fit-coords (type word brd seed)

(if (< type 2) ; horizontal placement

(let\* ((opn (open-brd-coords 0 brd)) ;opn-brd\_>final-coords->wd-fits->plc-cord

(mf (coords-horiz 0 opn))

(wd-fits (do-fits word mf))

(rcords (rand-coord seed (remove nil wd-fits)))

(ret-cords (rand-start (car rcords) word seed type)))

ret-cords)

(let\* ((opn (open-brd-coords 0 brd)) ; else vertical placement

(mf (coords-vert 0 (openv 0 opn)))

(wd-fits (do-fits-vert word mf))

(rcords (rand-coord seed (remove nil wd-fits)))

(ret-cords (rand-start (car rcords) word seed type)))

ret-cords)))

;-------------------------------------------------------------------

;---------Place these words in the Board----------------------------

;-------------------------------------------------------------------

#|

; Place this word vertically dog

(defun plc-vert (brd word coords)

(let\* ((col-num (cadar coords))

(y1 (caar coords))

(new-brd (col-rep brd word y1 col-num )))

new-brd))

;Find a place letters horizontally

;across the game-board this will do it

(defun plc-horiz (brd word coords)

(let\* ((row-num (caar coords))

(y1 (cadar coords)) ; y1 value for placement

(new-brd (row-rep brd word row-num y1)))

new-brd)) ; return new board |#

#|======================================================================|#

#| START OF DIJKSTRA CODE |#

#|======================================================================|#

; (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)

(if (verify-placement word brd column row "right")

(replace-characters word brd column row "right")

brd))

; Left placement

((= type 1)

(if (verify-placement word brd column row "left")

(replace-characters word brd column row "left")

brd))

; Down placement

((= type 2)

(if (verify-placement word brd column row "down")

(replace-characters word brd column row "down")

brd))

; Up placement

((= type 3)

(if (verify-placement word brd column row "up")

(replace-characters word brd column row "up")

brd))

; Right-Down placement

((= type 4)

(if (verify-placement word brd column row "right-down")

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

brd))

; Left-Down placement

((= type 5)

(if (verify-placement word brd column row "left-down")

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

brd))

; Right-Up placement

((= type 6)

(if (verify-placement word brd column row "right-up")

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

brd))

; Left-Up placement

((= type 7)

(if (verify-placement word brd column row "left-up")

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

brd)))))

; (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))))))

; (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 (cdr brd))

(type (rand 8 seed))

(new-seed (next-seed seed))

(x-coord (rand 12 new-seed))

(another-seed (next-seed seed))

(y-coord (rand 12 another-seed))

(start-coord (list x-coord y-coord))

(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 (append solutions (list new-word-solution)))

(new-brd (cons new-board new-solutions))

)

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

;(plc-wdsrch (list "hello" "test" "world")

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

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

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

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

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

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

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

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

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

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

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

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

; (list #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\. #\.)) '()) 78)

; Old unused code

#|

(defun plc-wdsrch (words brd seed)

(if (endp words)

'()

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

;(type (rand 4 seed)) ;get the type we are placing

(type (rand 8 seed))

(coords (fit-coords type word brd (+ seed 57)))

(new-brd (place (car brd) word type coords)));our new updated board

(if (or (= type 1) (= type 3) (= type 6) (= type 7))

(cons (cons (cons (car words) (reverse coords)) new-brd) (plc-wdsrch (cdr words) new-brd (+ 99 seed)))

(cons (cons (cons (car words) coords) new-brd) (plc-wdsrch (cdr words) new-brd (+ 39 seed)))))))

|#

;-------------------------------------------------------End Placement

;-------------------------------------------------------Unused

;; Changing values of a row puts the

;; correct spot utilizing coordinates

;(defun row-rep (chrs y1 y2 cnt row)

; (if (= cnt (len row))

; (if (null chrs) '()

; (list (car chrs)));we reached end of row

; (if (and (not (endp chrs)) (and (>= cnt y1) (< cnt y2)))

; (cons (car chrs) ; where we put characters into board

; (row-rep (cdr chrs) y1 y2 (+ 1 cnt) row))

; (cons (nth cnt row) ; kee going until we are in range

; (row-rep chrs y1 y2 (+ 1 cnt) row)

;

;))))

;(defun replace-col (brd col col-num)

; (if (endp col) brd

; (cons (row-rep col col-num (+ 1 col-num) 0 (car brd))

; (replace-col (cdr brd) (cdr col) col-num))))

;End placement.lisp