Language Battles

"Come back with your shield - or on it" (Plutarch, Moralia 241)

Prizes[1] for top three finishers!

NOTE: Revision 1 11/15.

We will write a program that will be executed in a "gaming harness" that allows your program to play against other student's programs. Your program searches for a move to make within a given timeout period agreed upon by the two combatants and entered as a command line parameter when the match is initialized. If your program has not generated a move by the end of the time allotted you can make a pass move. No points will be accumulated for a pass move. If your program does not make either a valid move or a pass within the allotted interval you lose the match. If your program generates an invalid move you lose the match. [2]

You are given a board and a dictionary file (possibly more than one form of dictionary in different files). You can think of the game board as a three dimensional array of characters. The dimensions of the board are given as command line parameters to your program. A new board is generated for each match by a program I will provide. In order to make a move you need to find a sequence of contiguous characters along a row, column, or into the third dimension of the board such that the characters can be permuted to form a word contained in the dictionary provided. The word can be any length from 3 characters in length to the maximum for the given dimension. You make your move by writing out a line to standard out conforming to a format specification. A referee program that I will provide will receive your output across the network on another computer. Everything to make the messages traverse through the various computers will be provided for you. Your program only needs to read from standard in and write to standard out. Code to determine if input is available to be read from standard input will be provided to you. The code you provide need only concentrate of the logic of finding good moves not on the mechanics of transmitting the data to the opponent or getting data from the opponent. The moves you issue will be verified by the referee program. You will receive points for the word you found. When your move is made part of the information you will send will be an indication as to what positions (x,y,z begin x,y,z end) the letters came from. The letters at those positions will be removed from the board . Any vacancies created by removing those letters will be filled by moving letters up from the third dimension letters below the holes created. When all letters for a given x,y have been "bubbled up" from below a "blank" square fills in the void.

We will use the standard point values for letters used in the game Scrabble.

A 1	B 3	C 3	D 2	E 1	F 4	G 2	H 4	I 1	J 8
K 5	L 1	M 3	N 1	O 1	P 3	Q 10	R 1	S 1	T 1
U 1	V 4	W 4	X 8	Y 4	Z 10				

Obviously, the longer the word you submit the more points it is worth but you also need to consider the composition of the word as the value of the letters varies. The sum of all the point values for all the

1 Cheap

2 Invalid moves include: Coordinates don't define a contiguous block of letters according to the rules; Word submitted is not in the dictionary; Word is less than 3 characters in length; Characters denoted by coordinates do not for a permutation of the word submitted; time limit exceeded. All invalid moves result in loss of the battle.

letters in the word in your move is the score for that move. The total score of all moves is your score for the battle. Points are awarded in the tournament as follows: win 2, tie 1, loss 0. Thus, the total points issued for each battle is always 2. We *could* chose to have each player play every other player in the tournament. There are 34 players. That would mean each person would need to arrange for 33 matches. This is too much. Therefore, we will utilize brackets and a series of rounds with elimination.

There are 34 players so we will divide the class into four groups with 8, 8, 9, and 9 players respectively. Each player will play every other player in their initial (round one) group. The *top four* from each original group will progress to the second round. The second round consists of 2 groups of 8 players. Each player plays all other players in their group of 8. The *top four* from each group of 8 progresses to the third round. The third round has one group of 8. Each player plays all other 7 players. The *top four* advance to the fourth round (semi-finals). Each player in the semi-finals plays three other players. The *top two* progress to the title match. The other two have a play-off to determine who gets third place and fourth place. The winner of the title match is the first place finisher, the loser is the second place finisher. The *top three* positions get a really worthless cheap prize, to be determined later, and all the concomitant glory.

In the event of ties *in any round* that makes the determination of who advances not determined by the normal number of matches for the round, there will be a play-off match or matches as needed to determine who advances. The tournament consists of 5 rounds. I propose we hold one round on each of the last five days of the term, **Monday through Friday of dead week**. Therefore, you need to have your program ready to go for the first round of play by Monday **November 28th** the start of "dead week." It is permissible (encouraged) to keep refining your program as the tournament proceeds. The tournament will place people into 5 result groups for the purpose of assigning a grade for the project.

Position	Grade
Final Four	?
Eliminated in Round 3	??
Eliminated in Round 2	???
Eliminated in Round 1	????
Program Doesn't Work Can't Play the Game	?????

Making the brackets

I will provide each student in the class with an *alias code*, a unique number. You will be identified on the bracket chart by that number. You will receive an initial assignment into a round one group. I will provide you with a list of email addresses for the other members of your group. You will contact the other members of your round one group and arrange to play a match. You will not know the alias code for any of the other players. The results will be placed on the bracket chart by my office door using the alias codes. If you are extremely shy and desire *complete anonymity*, let me know and I will arrange to meet your match opponents and play your code against them for you. As matches are completed the results will be posted on the bracket chart using the alias codes.

The letters in the example below should all be in lower case but the word processor is fighting me trying to upper case them. Just pretend they are all lower case. The following example shows a 6x6x3 playing board. The first 6x6 board is the first plane in the third dimension etc. A possible move could be 1,1,1 1,5,1 forming the word APPLE. This would score 9 points. Another possible move would be 1,3,1 3,3,1 forming the word TOP as PTO is a permutation of TOP. This would score 5 points. When the APPLE move was registered the letters OARDT would bubble up from the 2nd plane in the 3rd dimension to fill the positions vacated by the letters in the positions occupied by APPLE on the 1st plane in the 3rd dimension. The letters GENER would bubble up from the 3rd plane in the 3rd dimension to fill the slots in the 2nd plane of the 3rd dimension vacated by OARDT. The position in the 3rd plane in the 3rd dimension vacated by the letters GENER would be fill with blank markers. The first letter for a move always begins with a letter from the 1st plane of the third dimension. It can take letters from a row, a column, or it can burrow down through corresponding positions of the planes in the 3rd dimension. For example 6,3,1 6,3,3 LOT would be a legitimate move. When you bump up against a *space character* in a cell you can't proceed further in that direction. For example, if the LOT move had been made you couldn't form the word BUT from 6,3,2 to 6,6,2 because you would bump into the space at 6,4,2 where the O from LOT used to be.

A	P	P	L	E	B
U	T	T	E	R	I
S	G	O	O	D	T
O	E	A	T	T	H
I	S	I	S	A	S
A	M	P	L	E	B

O	A	R	D	T	H
I	S	G	A	M	E
W	I	L	L	F	O
R	C	E	Y	O	U
T	O	T	H	I	N
K	A	B	O	U	T

G	E	N	E	R	A
T	I	N	G	P	E
R	M	U	T	I	O
N	S	A	S	T	O
P	P	I	N	G	A
N	D	S	T	A	R

Hopefully, that explains how the board is managed and what constitutes a legitimate move.

Provided Code

I will provide the code you will need to insert into your program to deal with checking to see if input is available (meaning you need to start your move timer). I will provide the code for all the routines needed to deal with *time* and how your program will know it needs to submit its move NOW before the move interval expires. This way you can just concentrate on the logic of how to find moves. If you are trying to get a start on your code before I have completed the sample code I will be providing just put in a dummy stub routine for something that you call to check to see if input is available. Have your stub routine indicate no input is available a few times and the indicate there is input available and then read from standard in to manually type in a move to your program simulating that the other player made a move. Have the stub just behave like this in a loop forever. This way you can go ahead and do all your development and debugging before actually having the code I am writing for you.

Move and Pass Message Formats

The make a move you just write this out to standard out (a Unicon write is all you need).

W,3,2,1,3,5,1,help

This would be a move from the White Player (W means white player, use B if you are playing black). It indicates that the word starts at coordinates 3,2,1 (x,y,z) meaning 3rd row 2nd column top plane and ends at coordinates 3,5,1. Hence, the word is 4 characters long along row 3 of the top plane of the board. The word you are asserting to be a permutation of the characters at those positions is "help." That is all there is to it. When testing your program just manually type something like this into standard in to simulate the opponents move. Of course, it should actually make sense for the board you are using. You should think of the s,y,z as representing the row, column, and plane.

The make a pass move you write out
W,P or B,P depending on what player you are.

Board Format

The board will be generated uniquely for each match. It will be placed into a file named board. Since you will know the dimensions of the board to be used in this match from command line parameters given to your program there is no need to imbed that information into the file. Nonetheless, just as a sort of sanity check to make sure you have made the right board file available to your program I will place the x,y,z dimensions in the board file as the first line in the form x y z, meaning rows, columns, planes. For example, for a 10 x 20 x 30 board, the first line of the board file would look like
10 20 30

This means 30 planes each consisting of 10 rows with 20 characters per row.

Following the first line of the file the characters for the board will appear one row per line. The order will be to give row 1 through n for the 1st plane in row major order. The will be followed by the rows for the 2nd plane as so forth until all planes have been filled. The board from the example above would

appear in a file as the following series of lines.

```
appleb
utteri
sgoodt
oeatth
isisas
ampleb
oardth
isgame
willfo
rceyou
tothin
kabout
genera
tingpe
rmutio
nsasto
ppinga
ndstar
```

Dealing with the Passage of Time

You program needs to be aware of the passage of time in order to make your move before the timeout has expired. I had hoped to give you some code that would utilize the real time clock timer associated with your processes. I wrote up the code to do this but unfortunately, I seem to have uncovered a bug in the implementation of the virtual machine related to returning from traps that catch signals. The code works properly but only for an indeterminate number of cycles of the interval timer. If it runs long enough it will trip across the flaw in the run-time system implementation. I have been working with the developers to try to locate and fix this issue. This type of bug that only manifests itself very infrequently and that involves the very complicated establishment of traps that are ultimately caught in the C code of the run-time system and made to appear that they are serviced by procedures executing on the virtual machine are tricky to nail down. I do not anticipate that we will be able to isolate this bug and fix it in time for our project's use. Therefore, we are going with Plan B.

The sample code I gave you in the step by step instructions for how to install the compiler shows how you can get from the operating system its conception of the current time expressed as two numbers. These two numbers represent the number of seconds and microseconds that the operating system believes have transpired since midnight January 1st , 1970. We can simply uses this aas a way of keeping track of the passage of time. Rather then an *interrupt driven* approach this is more of a *polling approach*. Frankly, it is less elegant than the original approach I wanted to provide for you but its what we can get working in the time we have.

You need to write you code so that at various places in your program you call a procedure to update how much time has passed since you cause the current interval to be reset. When you are searching the board for possible moves you need to do it in incremental chunks and check in with the time to see how much time you have left before you need to send a move. You can implement you own time related procedures in your program if you wish. The procedures I have provided here are sufficient for your needs. The built-in Unicon procedure *gettimeofday* returns a record with two fields, sec and usec.

Notice how *gettimeofday* is naturally a generator. These are the seconds and microseconds of the *epoch* respectively. We will need a procedure to reset to the beginning of an interval and a procedure to tell us how long it has been since we did a reset.

I have written two procedures and put them into a package called timefuncs that you can use to keep track of the passage of time. They are on Canvas under Files-→Language Project. The timefuncs.icn is the package containing the procedures time_reset and time_passed. They should be rather self-explanatory. When you want to initiate a new interval of time to track call time_reset. Until you call time_reset again the time_passed procedure will tell you how long it has been since the last reset. This should be all you need to keep track of time passing as your program executes. You will need to fine tune you program by placing some calls to passed_time around at various places in your program and write out how much time has passed so you can get a feel for how long it takes various pieces of you search to execute. This will enable you to decide what further searching you have time for as the move time interval is running low. You want to try to let your program search as deep as it can in order to find a move that will bring a good score but you don't want to cut it too close or your move will not be registered in time and you will lose that match.