Five Crowns LISP Manual

Author: Frank Ryan

Submitted: 10/22/2019

Bug Report:

Loading invalid load file, and saving invalid directories.

Feature Report:

    Missing features: Validation of load file

                Validation of saving file

Shuffling a deck

    Extra features: none

Description of data structures:

To run my program, the main function to call is the beginGame function. The function takes no parameters, and is to be called as … (beginGame). This function will call the mainScreen function, beginning the program.

    Lists are used for most data in my program. The player hand, draw pile, and discard pile are all lists. In some functions, such as goOutTest1 and checkRun/Book, I use multiple lists within one list. The input from the human player are recognized as atoms. The round, computer points, and human points are also recognized as atoms. I use many LISP functionalities to create new data, such as (first list) and (rest list), I never use any destructive functions.

    Primary Function Descriptions:

    checkRun1, checkRun2, checkRun3: These functions are called to find all runs from a card list. The checkRun3 finds all runs of a certain card for a certain size. The checkRun2 function stores the run found in checkRun3 for the card of that size, and then looks for a run of a different size of THAT SAME CARD. The checkRun1 stores all runs found in checkRun2, and is then recursively called to get the next card of the hand, and pass that to checkRun2 to get the runs for that card.

    checkBook1, checkBook2, checkBook3: See checkRun1, checkRun2, and checkRun3 description. The functions are the same, except the functions are looking for books instead of runs.

    goOutTest1: Used to see if a player can go out. The parameters contain a player’s hand, a list of all his runs and books, and the minimum amount of cards left over from books and runs. Passes the first of the list of books and runs to the goOutTest2 function. This function will return a list of books and runs from the hand. The first of this returned list will be the leftover cards that are not used in books/runs. A condition will see if the length of the first of this list is less than the minimum length parameter. If it is, goOutTest1 is called again with the rest of booksAndruns when it was first called, the same hand, and the new minimum, which is the returned list of goOutTest2. If the length of the returned list is not shorter, then goOutTest1 is called again with the same hand and minimum, but also with the rest of booksAndruns. When booksAndRuns is empty, the best combination of books and runs with the least amount of leftover cards should be returned.

    goOutTest2: When called from goOutTest1, the parameters it has is the book/run passed, and the hand that does not contain the cards in the passed book/run. This function then creates a new list of runs/books with the leftover hand. It is then called again with the first book/run of this new list, and new hand that removes this first book. The base case for this function is when there can be no books/run created from the hand. It then returns all books/runs that were created, and the leftover cards in the hand, if any.

Log:

Milestone #1 - 10/8/2019

**From 10/1/19 to 10/5/19**… For the first couple of days, I tried to familiarize myself with the functionalities of LISP. I also had to understand how LISP works in terms of functions and recursion. Once I made some example code, I started working on the project to get a base main screen working for the game. From this, I learned how to handle input validation, as well as how to structure my project. After the main screen function, I went onto working on starting a new game, implementing the coin toss and determining a winner. From this I had to decide how to iterate through the rounds using recursion, but also keeping track of the players’ points.

I’ve noticed that I’ve had more success structuring code by working from the inside out of function, by determining what my final return data should be in function. As an example, when looking at players’ points, I realized that I needed a function to return the total points AFTER iterating through all rounds.

So after these days, I now have a base game working, where a new game can go through all rounds and the human player has all functionalities besides going out. I spent about an average of 5-6 hours on this.

**From 10/5/19 to 10/8/19** … Now that I have the base game done, I now have to implement going out function. Once I have this function done, which involves creating books and runs, I believe the rest of the project will be quite straightforward. However, this function will take the most amount of time, as I have run into a lot of problems with it.

The main issues about creating my go out function is determining books and runs, and placing ALL of them inside a list. Unlike the C++ project, I have no Card object to look at the value of the card and suite SEPERATLY. This has been very challenging, and one solution I have been thinking about is translating all cards of the deck into numbers, 1,2,3,4…122. This would allow be to compare the cards efficiently, looking at the card+1 for runs and card mod value for books.

Once I have functions to create all books and runs, I can then look at the actual goOut function by passing this list and recursively calling it, similar to how an example was presented in class. I have spent around 5 hours throughout these days.

ANOTHER PROBLEM: I have also not found a way to shuffle the cards in the deck. I’ve looked online and other places, but it seems that I will have to find my own way to shuffle the cards, rather than just a simple function like in C++.

Milestone #2 - 10/15/2019

**From 10/9 to 10/12 …**

I’ve finished creating the algorithms to find all books and runs (however, I have not dealt with wilds yet). These algorithms took a long time to structure, but once I figured out a way to structure my functions, the steps of each function were simple. From this, I was able to find a list of all books and runs in a player’s hand.

Moving on, I looked to create the function to check if a user goes out. I based my algorithm around the one that was suggested in class. However, at first, it took me a while to create the “tree” of possibilities, as I kept getting a stack overflow error (from bad recursion).

Because of this, I had to restructure my function. After creating another function to stop this stack overflow, my goOut function was complete.

During this time, I would say I spent an overall of 15 hours here.

**From 10/12 to 10/15 …**

Now that most of my essential functions were done, I now had to create functions to handle wilds and also count card points.

In my checkRun and checkBook functions, I created a conversion a player’s hand that converts all face cards to their actual value, ex. 10C for XC. So, in order to add up cards such as these, I had to convert the hand back to face values and then count the cards.

However, when creating functions to handle wilds, I had many problems. When looking at jokers, I needed a value that would work inside my checkRun and checkBook functions. If I were to translate jokers to 50, then my functions would break. So, instead, I changed jokers to have number value 15. I had to do this because my functions would try to add the second number value by 10 and check what value it is. If a joker were to be 50, then adding 10 to the second number would equal 10. Along with this, if I added 10 to the second number of an X card, then it would also equal 10. So, because of this, I made jokers equal 15.

After successfully handling wilds and creating functions to count card points, I am now ready to create logic to the computer. Even though I just started the computer now, the computer’s logic will be entirely based off of the goOut function, so it should be quite simple moving forward. Once this is done, I will transfer this logic over to the human’s ask for help and finally, I can move onto serialization.

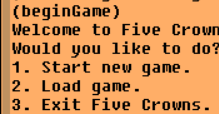
Overall, I would say I spent about 10 hours here as well.

NOTE: I’ve ran into an error during serialization, where Allegro can’t find the directory of my txt file. I’ve tried multiple ways to refer to directories, but I’ll have to keep looking.

To run my program, the main function to call is the beginGame function. The function takes no parameters, and is to be called as … (beginGame). This function will call the mainScreen function, beginning the program.

Screen Shots:

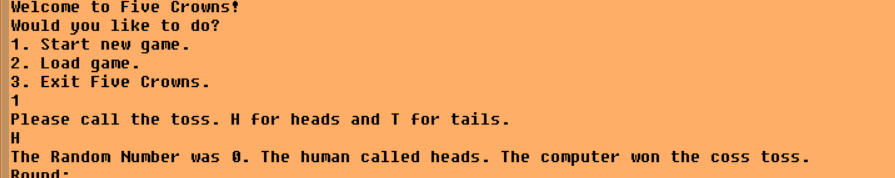
Main function to call to begin game –



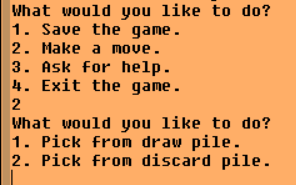
Base Game on the screen –



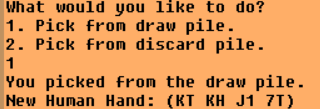
Human Input for Beginning Game and Coin Toss –



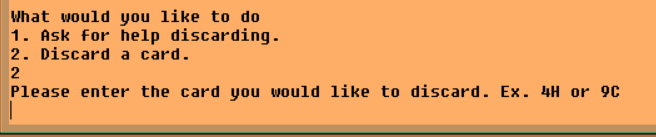
Human Choosing to Make a Move –



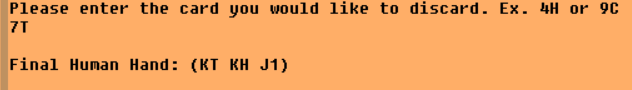
Human Choosing to Pick from Draw or Pick from Discard –



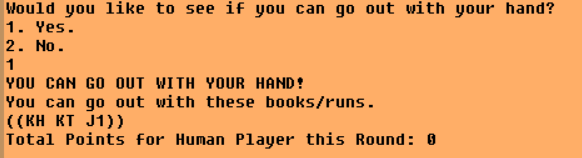
Human to Discard a card –



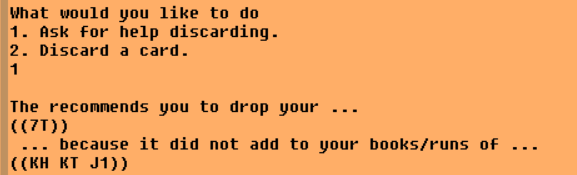
Human Discarding a Card –



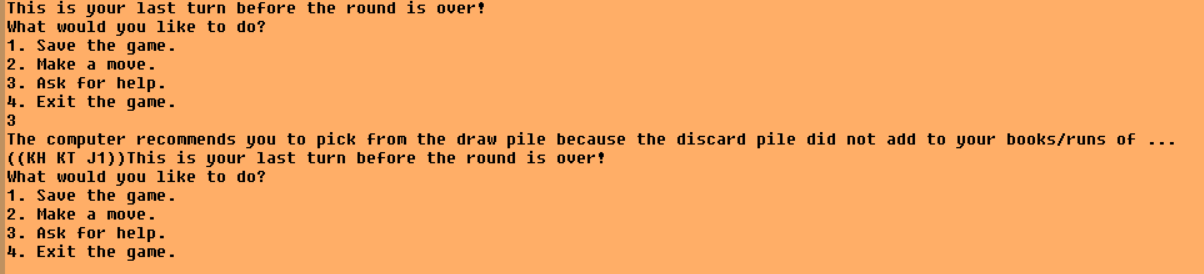
Human Choosing to Try to Go Out with Hand –



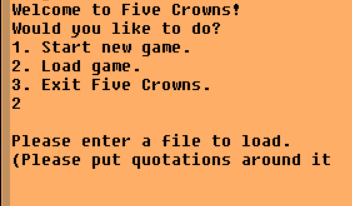
Human Asking for Help to Discard –



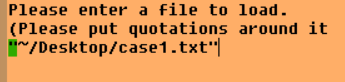
Human Asking for Help to Pick Card –



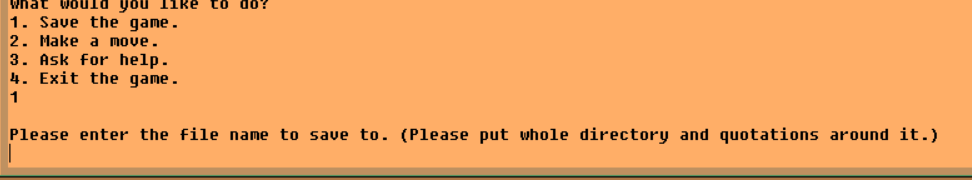
Human Choosing to Load a Game from file –



Human Entering a file to Load from (MUST USE QUOTATIONS AROUND FILE) –



Human Choosing to Save Game –



Human Entering File Name to Save to (MUST USE QUOTATIONS AROUND FILE) –

