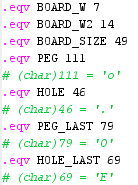
# Implementation

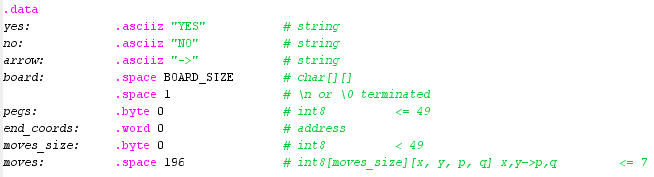
Peg solitaire can be solved by a backtracking algorithm because it involves using a permutation of moves to reach an end goal. The process will be discussed along with the functions used in the following sections.

## Constants



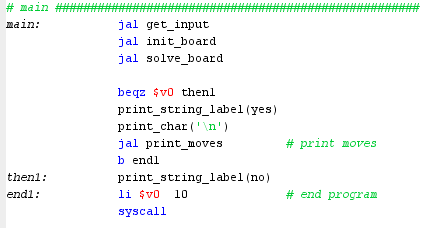
These are just constants used to make the code more robust to changes. The only thing to note here is that *BOARD*\_W2 is just two times the *BOARD*\_W.

## Global Variables



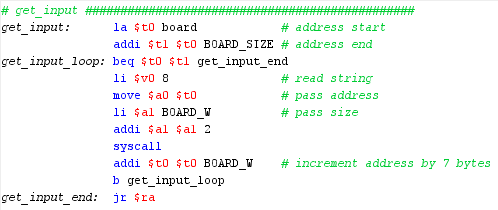
*pegs* stores the number of pegs currently in the board  
*end\_coords* stores the address of the position where the last peg should reside  
*moves\_size­* stores the number of moves taken to reach the end goal if possible  
*moves* stores the moves taken in reverse order

## Main



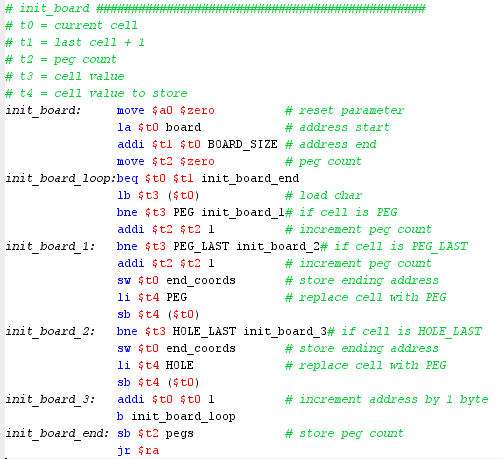
We can see the overview of the program in the main. The *solve\_board* function returns 1 or 0. If it is 1, it means that the board is solvable thus we print the moves; else, it is not solvable.

## Input handling



Input is handled by taking the input line by line while saving it directly to the *board* global variable. Notice that it doesn’t save registers to the stack. This is because this function does not use any preserved variables.

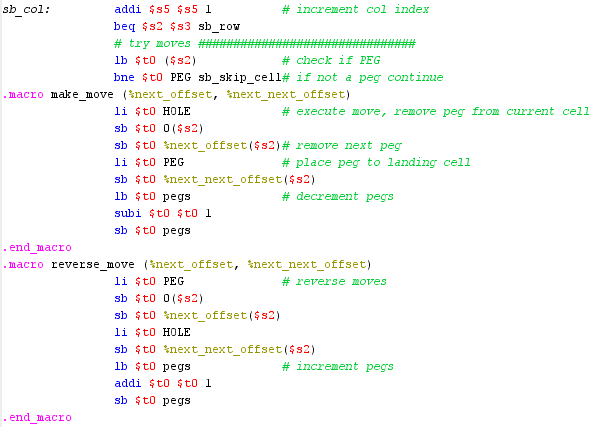
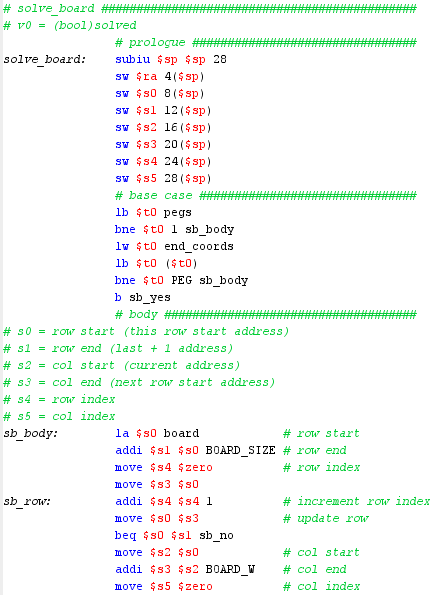
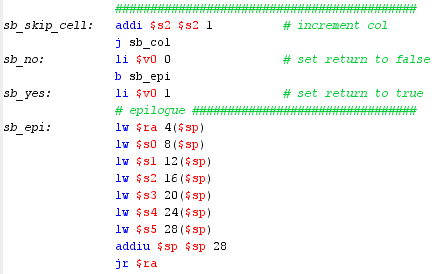
## Initializing the board



It initializes the board by setting the *pegs* global variable to the right number of pegs. It also checks where the last cell should be (i.e. where the ‘E’ or ‘O’ is) and stores the address of this cell to the *end\_coords­* global variable. Lastly, it also replaces the ‘E’ with ‘.’ and the ‘O’ with ‘o’ to make the board consistent since we already saved the address of either ‘E’ or ‘O’. Notice that it doesn’t save registers to the stack. This is because this function does not use any preserved variables.

## Solving the board

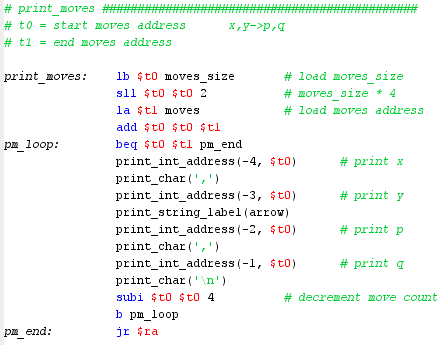
The code for solving the board is too long to include here. Thus, trivial parts are removed and replaced with a short description of what it does. Hopefully, this makes the code a lot more readable.



for each direction:  
 if move is valid:  
 do move  
 recursive call to *solve\_board*  
 if *solve\_board­* returns true:  
 save move made in *moves* array  
 increment *moves\_size*  
 return true  
 else:   
 reverse move   
 else:   
 continue

The *solve\_board­* function is a simple backtracking algorithm. It iterates over each cell in the board and checks each direction if a move is possible. If it is, it does the move and a recursive call to *solve\_board*. It checks its return value. If it is true, it saves the move and return true also. Else, it reverses the move made and continue with the iteration over each cell. If there are no moves left, the function returns false to backtrack. Notice that with this algorithm, we save the moves in reverse order. This will have an effect to the *print\_moves* function.

## Printing the moves



To print the moves taken, we should first take note that the *moves* array is the sequence of moves taken in reverse order. Thus, when printing, we start from the end of the array by adding the *moves\_size* to the start address of our *moves* array. It then prints the moves accordingly line by line. Notice that it doesn’t save registers to the stack. This is because this function does not use any preserved variables.