Programming Lab 4: Minesweeper, Part 2

EE 306: Introduction to Computing
Professor: Dr. Al Cuevas
TAs: Apurv Narkhede, Nic Key, Ramya Raj, Jerry Yang

Due: 12/3/2018 at 12pm

1 Overview

This lab is intended to familiarize you with subroutines, linked lists, and output using TRAP (sub)routines. By the end of this lab, you should be able to:

1. write and debug up to 1000 (hopefully no more) lines of assembly code.
2. use modular programming to break big problems into smaller problems.
3. address issues with code that is too long.
4. use LC3 TRAP instructions for input/output operations.
5. create a functional game.

Late submissions will not be accepted; grades will be published within 1 week of the due date. Regrade requests will only be considered if the autograder is incorrect (see regrade policy below).

Note: You may NOT show anyone other than the TAs or Dr. Cuevas your code.
Any violation of this rule constitutes academic dishonesty.

2 Background

This lab is the second part of a two-part lab that will culminate in the game Minesweeper. You already know a little bit about how the game works: the player chooses spaces on a grid, using the number of bombs nearest to the space of choice to determine where the hidden bombs are. In the previous lab, you programmed most of the "back end" of the game, that is, the behind-the-scenes. In this lab, you will program the "front end" - taking user input, checking if the input is valid, updating the board, keeping score, and programming a game-over.

Most of the ideas and concepts for Lab 4 carry from Lab 3; the only new ideas are using modular programming and using TRAP instructions to take in keyboard input.

2.1 Modular Programming (and a bit of coding philosophy)

Programming is just as much a skill as it is a way of thinking and problem-solving. Programmers have to take very large, broad, and open-ended questions and somehow come up with a good solution that meets the requirements of all the stakeholders involved in process. A key component of developing a solution to a large problem is by breaking it up into smaller, more accomplishable problems and writing code to tackle those smaller problems before bringing it all together. This practice is called modular programming.

Subroutines are the key to good modular code. A subroutine allows you to solve a small problem, such as how to multiply two signed numbers or traverse a linked list, without needing to solve the entire problem. In Minesweeper Pt. 1, we asked you to write four specific subroutines integral to Minesweeper. This approach allowed us to break the larger problem of "How do we make Minesweeper in LC3?" into much more tangible and achievable steps - displaying the board, loading the board, and counting bombs. Had we simply told you to make Minesweeper, you would likely be quite lost. As in Lab 3, you are encouraged to write your own subroutines as necessary.

Modular programming is not just a coding style - it is a way of organizing and thinking through the tasks that you need to get done to accomplish your goal. As you become more competent in coding in general, you will encounter problems and issues that you won't know how to solve. Starting with what you know how to solve and thinking about different approaches to tackle what you don't know how to solve will place you in a position to view and use programming as a tool, rather than just a hobby or a skill.

This lab is intended to be open-ended to embrace both modular programming and creative thinking. While the lab document provides you with a sense of some of the problems you will need to tackle, it will not describe every hurdle you will meet. As the teaching team, we will not insist on a particular method or algorithm, but we will give suggestions and ideas that you may consider when implementing your solution. The only requirement is that your game has to work per the specifications mentioned in the lab document.

Have fun, and happy coding!

2.2 TRAPs and Console Input

Recall that TRAPs are special subroutines provided by LC3 that you can use to interact with the console. You will be using GETC and PUTS in this lab. Since you used PUTS in the last lab, we will only discuss GETC.

2.2.1 Taking in a Character as Input: GETC

GETC waits for a user to type a key from the keyboard onto the console. It stores the typed key as an ASCII character in R0. For example, if "0" is pressed on the keyboard, GETC will return, and R0 will contain x30. To use, write GETC in your code as a normal instruction without operands.

Note that GETC does not print the character it received. You will need to call OUT to print the character on the console. This is called "echoing" the character. An example:

```
; Echoes a typed character.
GETC
OUT
```

3 Lab Specifications

In this lab, you will implement five key subroutines that will build up to a functional Minesweeper game. All five subroutines will be tested individually to ensure you followed the LC3 Calling Convention (see section 2.2.2 of the Lab 3 document). You may copy and paste your subroutines from Lab 3 into Lab 4. You may also write your own subroutines (highly recommended!). Please see the Hints section for more details and the Expected Output section for a test run.

As before, do not modify the starter code given to you.

3.1 Subroutine 1: GET_MOVE

- Inputs: None.
- Outputs: R0 - Input 1; R1 - Input 2.
- Purpose: Prompts the player for two inputs that correspond to a move, echoes the player's keystrokes, and stores the move into R0 (row) and R1 (col).

A prompt is printed to the screen that updates as the user enters inputs:
"Enter a move: ("
"Enter a move: (2," ← here the user typed in 2 to the console
"Enter a move: (2,1)"← here the user typed in 1 to the console

What the user actually sees is: "Enter a move: (2,1)"

3.2 Subroutine 2: IS_VALID_MOVE

- Inputs: R0 - Input 1 (row of move); R1 - Input 2 (column of move).
- Outputs: If valid, set R0 - Row of desired move; R1 - Column of desired move. If invalid, print an error message to the console, and set R0 = -1.
- Purpose: Checks if the player's move is valid.

A valid move is a move that is within the bounds of the grid and does not include non-number characters. If the move is valid, the inputs are converted from ASCII to numbers between 0 and 3 inclusive, and (R0,R1) is set to the (row,col) of the move. If the move is invalid, the error message "Invalid move - try again!" is displayed, and a newline is printed to the console.

3.3 Subroutine 3: APPLY_MOVE

- Inputs: R0 - Row of desired move; R1 - Column of desired move.
- Outputs: R0 - Row of desired move; R1 - Column of desired move.
- Purpose: Checks if the move has been done before. If yes, print a message indicating so. If not, store the appropriate ASCII character (either a star or a number) into the board at GRID.

You will need a way to keep track of previous moves as well as a way to determine whether the location contains a bomb, a number, or a space (hint: you already wrote a subroutine that does this). If the player types in a move they have already done, print "You already tried this location!"

3.4 Subroutine 4: IS_GAME_OVER

- Inputs: R0 - Row of last move; R1 - Column of last move.
- Outputs: R0 - Status of game. 0 if the game is not over, 1 if the player won, or -1 if the player lost.
- Purpose: Checks if the game is over and if the player won.

If the last move revealed a bomb, the player lost. If there are no more possible moves remaining without revealing a bomb, the player won. If the move was not a bomb and there are still unchosen spaces in the grid, the game is not over. You may assume APPLY_MOVE is called before IS_GAME_OVER.

3.5 Subroutine 5: GAME_OVER

- Inputs: R0 - Denotes if the player won or lost. 1 if the player won, -1 if the player lost.
- Outputs: None.
- Purpose: Prints a winner/loser message, reveals the entire board, and prints the player's score.

If the player wins, display the message "Congrats, you won!". If the player loses, display "You lost! Better luck next time!". See the Expected Output section to determine newlines. The player's score should also be preceded by the string "Your score is: ". See the Expected Output section for newlines and formatting.

The score is defined as follows:

1. If the player wins, their score is 99.
2. If the player loses, their score is the number of attempted valid moves the player made during the game.

For example, if the player finds a bomb on the first try, their score is 1 (since they made 1 attempt). Note that the score can range from 0-16.

4 Expected Output

The Lab 4 folder on Canvas contains two screen recordings of me playing Minesweeper. The video titled "won" shows what should happen if I win the game. The video titled "lost" shows two cases in which I lose, and the expected output when I lose. The autograder expects you to match the output exactly as shown in the videos.

Here is a sequence of screenshots that shows Minesweeper in action. If the player wins, the loser message would be replaced by the winner message specified above. If the player has a double-digit score, it should print out as a double-digit number e.g. a score of 10 should be displayed as "10" on the console.

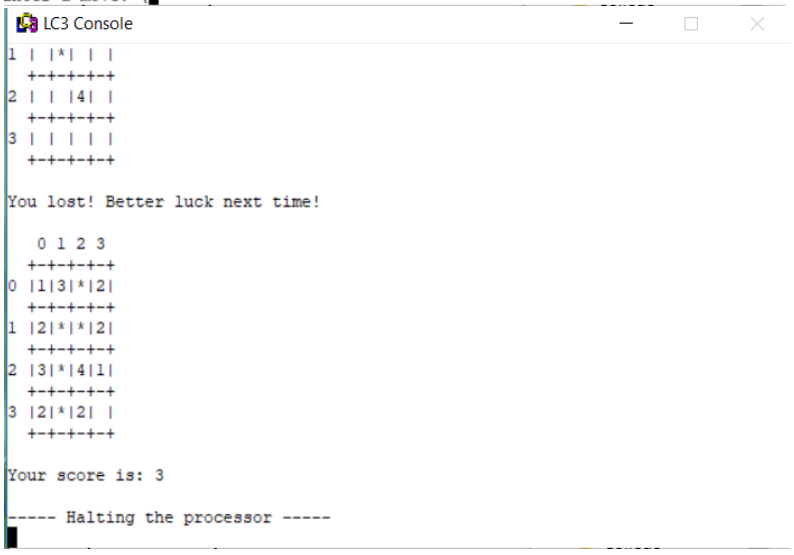

LC3 Console
Minesweeper
0 1 2 3
0
1
2
3
Board Initial
Enter a move: (


LC3 Console
Minesweeper
0 1 2 3
0
1
2
3
Board Initial
Enter a move: (2,


LC3 Console
0
1
2
3
Board Initial
Enter a move: (2,2)
0 1 2 3
0
1
2 |4|
3
Enter a move: (


LC3 Console
+-+-+-+-+
1 | | | | |
+-+-+-+-+
2 | | |4| |
+-+-+-+-+
3 | | | | |
+-+-+-+-+
Enter a move: (2,4)
Invalid move - try again!
Enter a move: (r,4)
Invalid move - try again!
Enter a move: (2,2)
You already tried this location!
0 1 2 3
+-+-+-+-+
0 | | | | |
+-+-+-+-+
1 | | | | |
+-+-+-+-+
2 | | |4| |
+-+-+-+-+
3 | | | | |
+-+-+-+-+
Enter a move: (


LC3 Console
You already tried this location!
0 1 2 3
+-+-+-+-+
0 | | | | |
+-+-+-+-+
1 | | | | |
+-+-+-+-+
2 | | |4| |
+-+-+-+-+
3 | | | | |
+-+-+-+-+
Enter a move: (0,0)
0 1 2 3
+-+-+-+-+
0 |1| | | |
+-+-+-+-+
1 | | | | |
+-+-+-+-+
2 | | |4| |
+-+-+-+-+
3 | | | | |
+-+-+-+-+
Enter a move: (


LC3 Console
1 | |*| | |
+-+-+-+-+
2 | | |4| |
+-+-+-+-+
3 | | | | |
+-+-+-+-+
You lost! Better luck next time!
0 1 2 3
+-+-+-+-+
0 |1|3|*|2|
+-+-+-+-+
1 |2|*|*|2|
+-+-+-+-+
2 |3|*|4|1|
+-+-+-+-+
3 |2|*|2| |
+-+-+-+-+
Your score is: 3
----- Halting the processor -----

5 Testing

1. We will test your subroutines for the LC3 calling convention!
2. We will test if you modified the starter code!
3. We won't use all of the linked lists we used to test Lab 3 as test cases for Lab 4. I suggest taking the autograder output from Lab 3 and creating your own linked list files.
4. If you get bored or sick of playing Minesweeper by yourself, get your friends to play it. Tell them to try to break your code; they usually get excited about that (at least my friends do).

6 Hints, Tips, and Tricks

1. You will need to devise a way to keep track of all previous moves. If there is a location that contains no bombs, the entry in memory will remain a space when APPLY_MOVE is called. However, there is no way to know if the player chose that location before. There are several approaches to this problem; my recommendation is to use an array or linked list to store previous moves.
2. You will also need to figure out how to keep track of the score. Again, there are several ways to do this.
3. Test each subroutine separately before putting it all together!
4. Come up with test cases and post them on Piazza!

7 Submission

1. Make sure you fill out the starter file header with your name, EID and recitation section time. Do NOT delete any information in the starter file, including comments.
2. Check your syntax to ensure it will work with the autograder. (See autograder syntax document on Canvas.)
3. Rename the Lab4.asm file "EIDLab4.asm", replacing "EID" with your EID.
4. Submit the Lab4.asm file (and ONLY the Lab4.asm file) to the Lab 4 Canvas assignment. Do NOT include any other files.
5. If you do not follow these submission instructions exactly, the autograder will spit out a 0!

8 Regrade Policy

We will only award points back if the **autograder** graded your code incorrectly, i.e. your code generated the correct output, but the autograder said you failed the test case. Unfortunately, we will NOT award points back if you fail all the test cases because of a single mistyped character.

If you wish to submit a regrade request, complete the following steps:

1. Review the autograder output, uploaded as a submission comment on your lab submission.
2. Visit Dr. Cuevas or a TA (preferably Jerry) in office hours to comfirm that the autograder output is incorrect **AND** that your code produces the correct output.
3. Email Jerry at jerryyang747@utexas.edu with the following:
    - Subject line: "EE 306: Lab 3 Regrade Request [EID]"
    - Screenshot of the autograder output
    - A description of what went wrong
    - Who you confirmed the error with
    - Your code, attached as an .asm file

After grades are released on Canvas, you have **7 days** to submit a regrade request.