ITP20002 – 01 Discrete Mathematics

Programing Assignment 1

Team 10

21700083 김도윤 21700632 장재성 21700486 윤하성 21700502 이민석

github address: <https://github.com/greentealemonade/DiscreteMaths>

Z3 must be installed for this program to work. (make build)

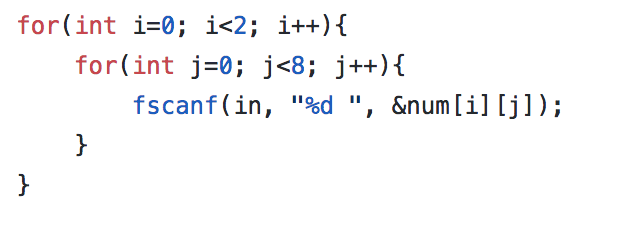
**Recurring Program Design**

File pointers below are responsible for each test file.

in - ‘input.txt’ form - ‘formula.txt’ middle - ‘middle.txt’ out - ‘output.txt’

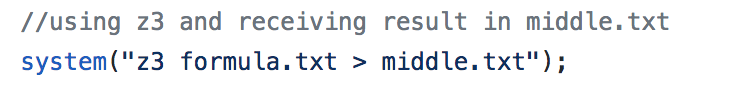
char **num** array is declared to store input values and char result array is declared to store final result values.

In every program, the input information should be read from ‘input.txt’. Since input formats are very consistent, command ‘fscanf()’ is used. ‘for()’ loop increases the convenience. The only adjustment needed for each puzzle case is the number of rows represented by ‘int i’ and number of columns represented by ‘int j’.

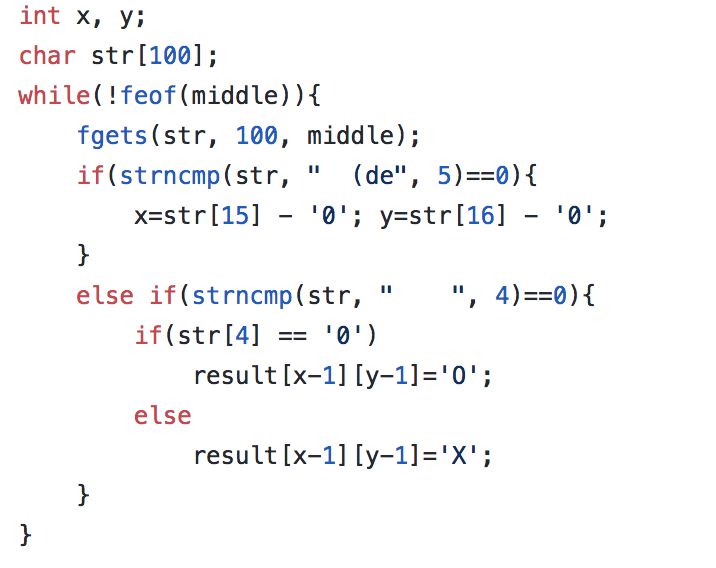
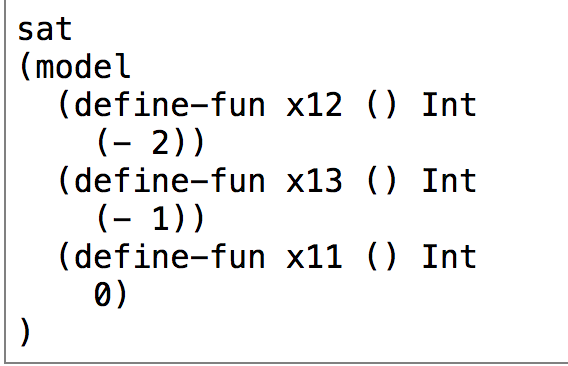


Then, each program implements its own logical formula in ‘formula.txt’, which will be described for each case far below.

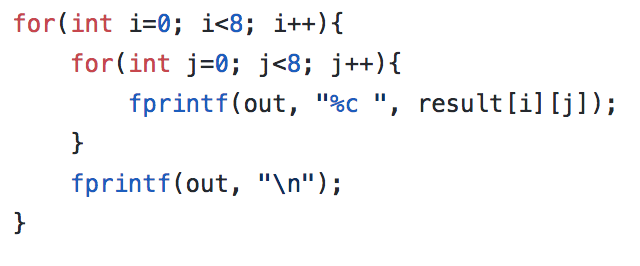
When ‘formula.txt’ is completed, system runs z3 program. Output of the z3 program is given to ‘middle.txt’. \**file pointer for ‘formula.txt’ must be closed before running z3 program.*



The output of the z3 program is given in such format on left bottom side. This output requires parsing. After ‘(model’part, the output is in a consistent format with ‘(define-fun x12’ part providing coordinates of the array and ‘(- 2))’ providing the value in the array(line 3 and line 4 as example).



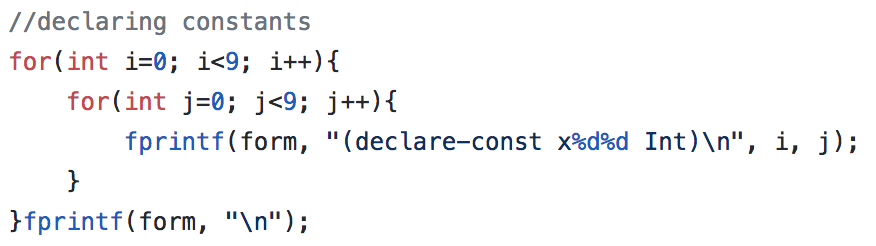
Output is printed to ‘output.txt’.



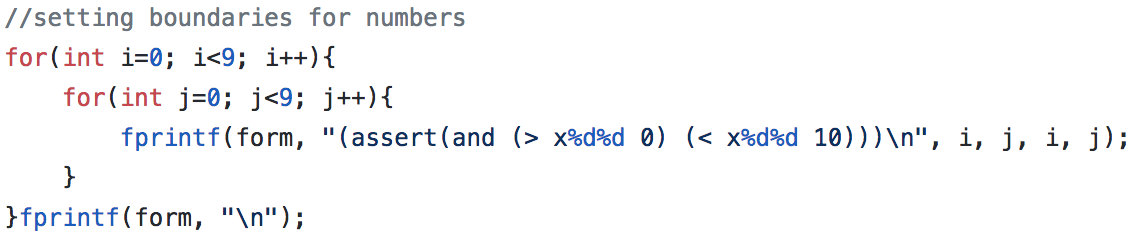
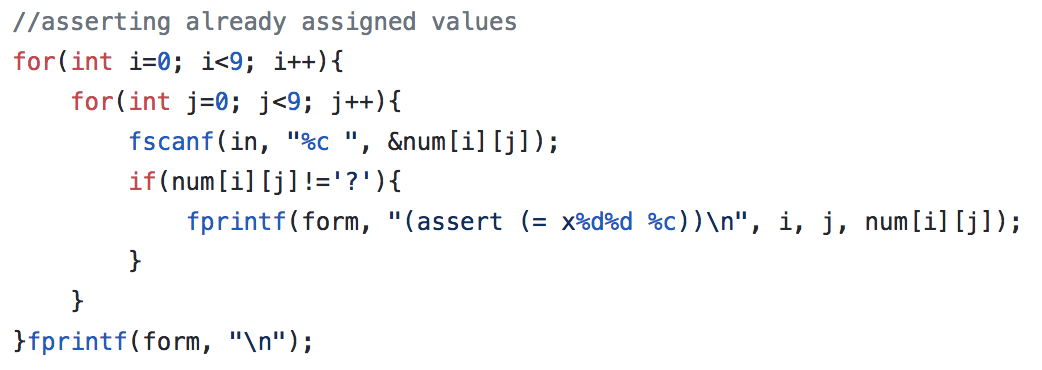
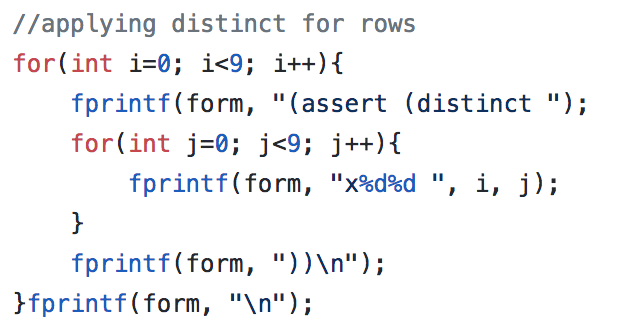
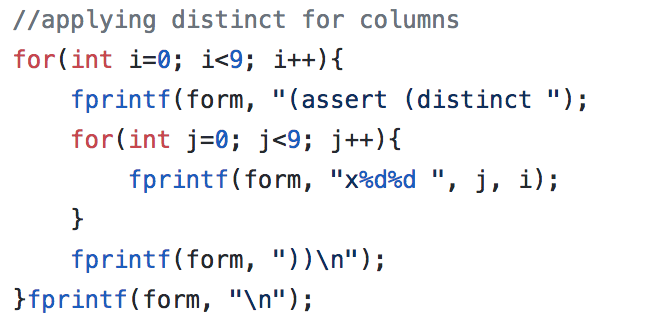
\*There was a lot of struggle running z3 in the program using the command ‘system(“z3 formula.txt > output.txt”). The command clearly worked when just typed on terminal, but didn’t work when it was contained within the c file. ‘output.txt’ file was created, but it was empty. This was because file ‘formula.txt’ was not closed before calling the system z3 by using the command ‘fclose(form)’.

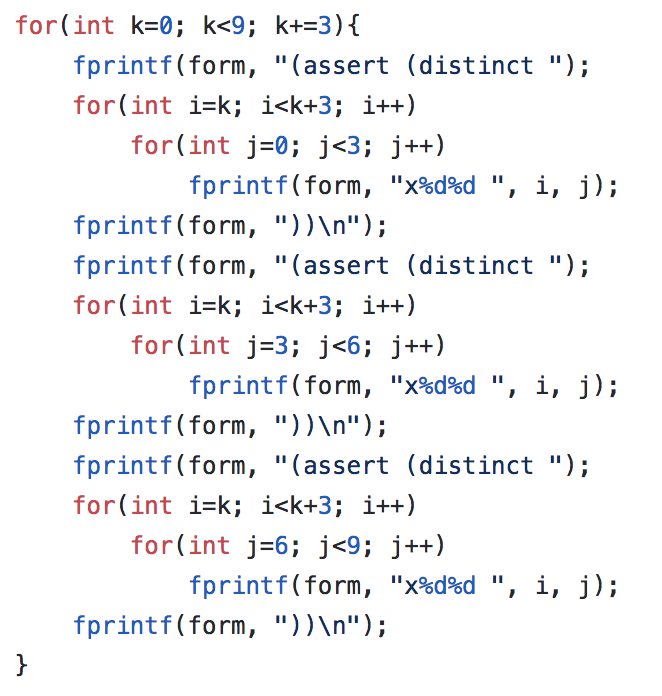
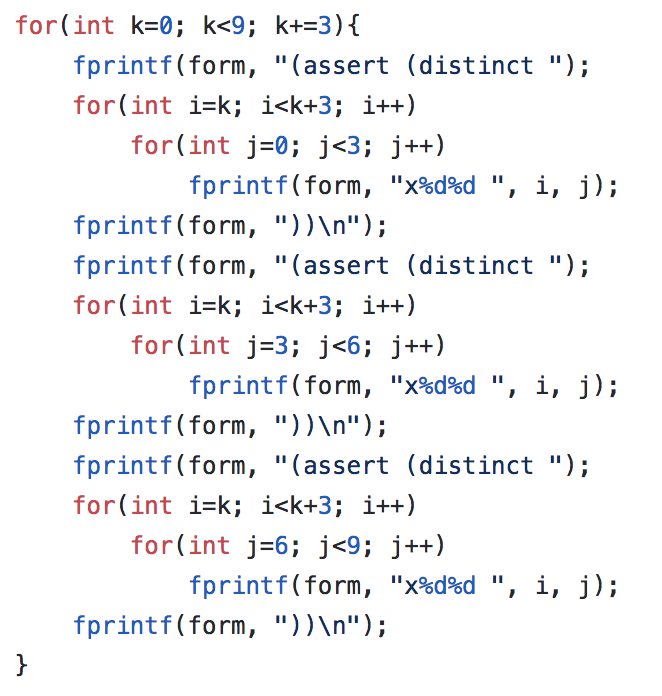
**Puzzle 1. Sudoku**

Constants are declared with the name from x00 to x88. Each digits representing the location in the array.

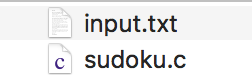


There are five types of assertions that needs to be done in sudoku. The example assertion will take first cell, first row, first column and first box as the example.

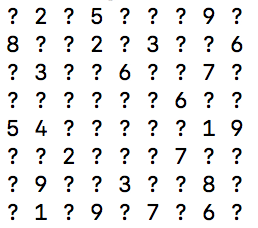
1. All numbers must be **from 1 to 9**
   * (assert(and (> x00 0) (< x00 10)))
   * 
2. Some numbers are already given
   * (assert (= x01 2))
   * 
3. every row must include numbers 1 to 9
   * (assert (distinct x00 x01 x02 x03 x04 x05 x06 x07 x08 ))
   * 
4. every column must contain 1 to 9
   * (assert (distinct x00 x10 x20 x30 x40 x50 x60 x70 x80 ))
   * 
5. every 3\*3 sudoku shape must contains 1 through 9
   * (assert (distinct x00 x01 x02 x10 x11 x12 x20 x21 x22 ))



**Manual:**



1. Prepare input.txt for sudoku.c. **‘input.txt’ must be in the same folder as ‘sudoku.c’**
   1. input.txt should contain the original sudoku problem in format below(9x9). Each number/symbol separated by space represents each cells in sudoku puzzle. ‘?’ represents blank cells in sudoku.





2. Compile sudoku.c file with the command below(in terminal/command prompt). ‘sudokusol’ is the name of the executable file, which can be set to different name.

* **gcc sudoku.c -o sudokuso**l



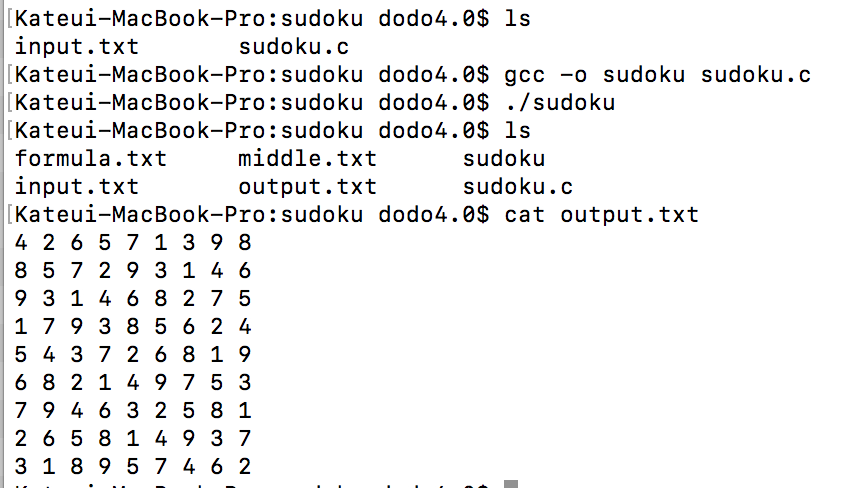
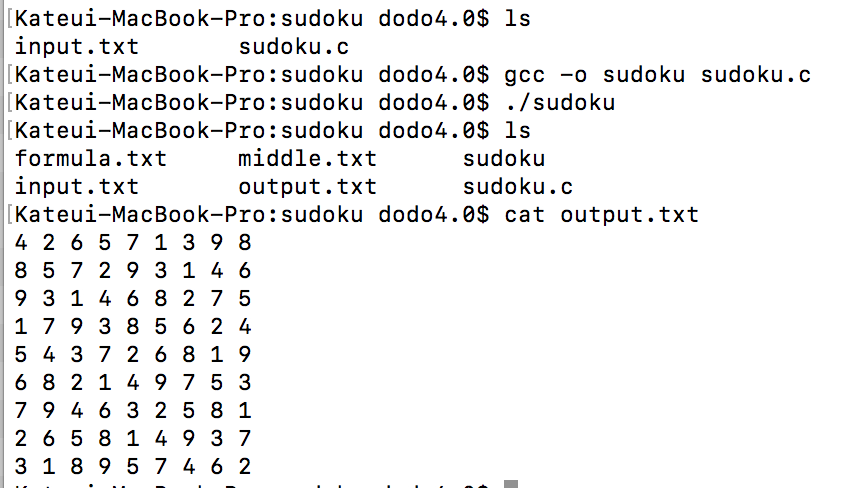
3. Run ‘sudokusol’ file. ‘sudokusol’ reads ‘input.txt’ file and writes ‘formula.txt’,

‘middle.txt’ and ‘output.txt’ files. ‘output.txt’ contains results

* **./sudokusol**

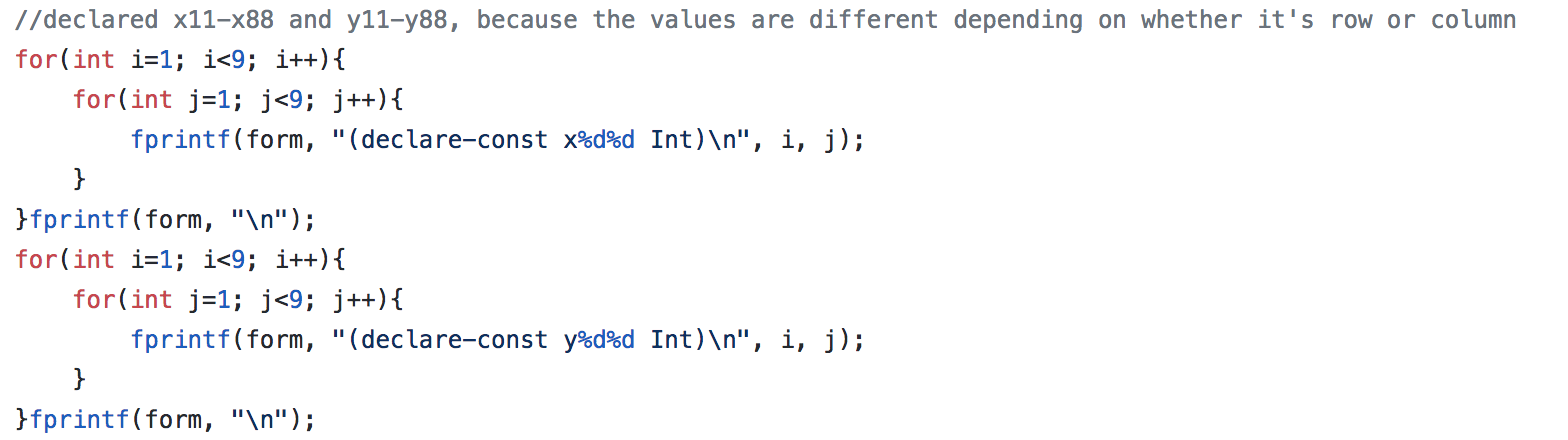
4. You can see the result in ‘output.txt’

■ Demonstration that shows your program works properly

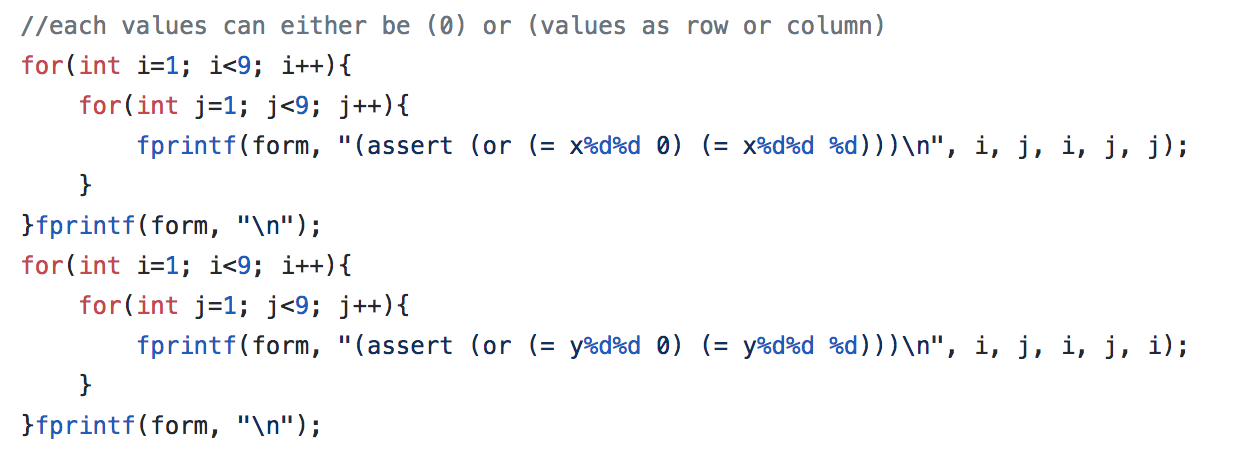
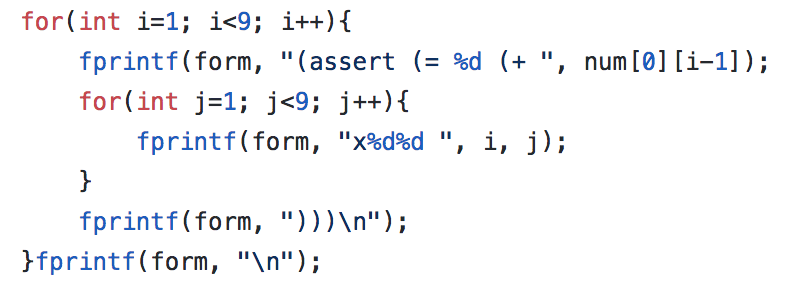
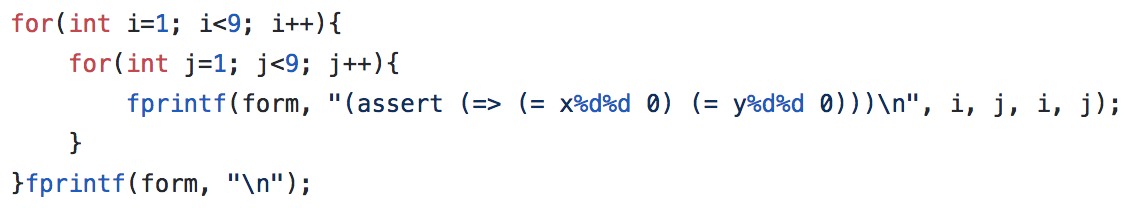


**Puzzle 2. Kakurasu**

Constants are declared with the name from x11 to x88 and y11 to y88. Each digits representing the location in the array. Because cells have different values to add to the sum, depending on its position as a row or column, two different constants were declared for each cell.

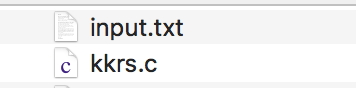


There are three types of assertions that needs to be done in kakurasu. The example assertion will take first cell, first row, first column and first box as the example.

1. Each cell is either activated or deactivated. If activated, it has a value of its **row integer or column integer**. If deactivated, it has **value of zero**. This is implemented by the usage of ‘or’.
   * (assert (or (= x11 0) (= x11 1)))
   * 
2. The input information earlier is the **sum** given for the sum of each row and columns. The program works to print structure of the formula below in ‘formula.txt’
   * (assert (= sum (+ x11 x12 x13 x14 x15 x16 x17 x18 ))) - row
   * (assert (= sum (+ y11 y21 y31 y41 y51 y61 y71 y81 ))) - column
   * 
3. It is important to make sure that when row integer is deactivated, column integer is deactivated too. ‘=>’ is a symbol for implication. It’s implements the relationship ‘if p, then q’.
   * (assert (=> (= x 0) (= y 0)) if x is 0, then y is 0 too.
   * 

When the result is 0, the program saves the result as O(deactivated) and when result is positive integer, the result is X(activated). The results are saved in char array result[8][8].

**Manual:**



1. Prepare ‘input.txt’ for ‘kkrs.c’. **‘input.txt’ must be in the same folder as ‘kkrs.c’**
   1. input.txt should contain the original kakurasu problem in format below(8x2). Each number/symbol separated by space represents sum of each rows and cols in kakurasu puzzle.





2. Compile ‘kkrs.c’ file.

* **gcc kkrs.c -o kkrssol**

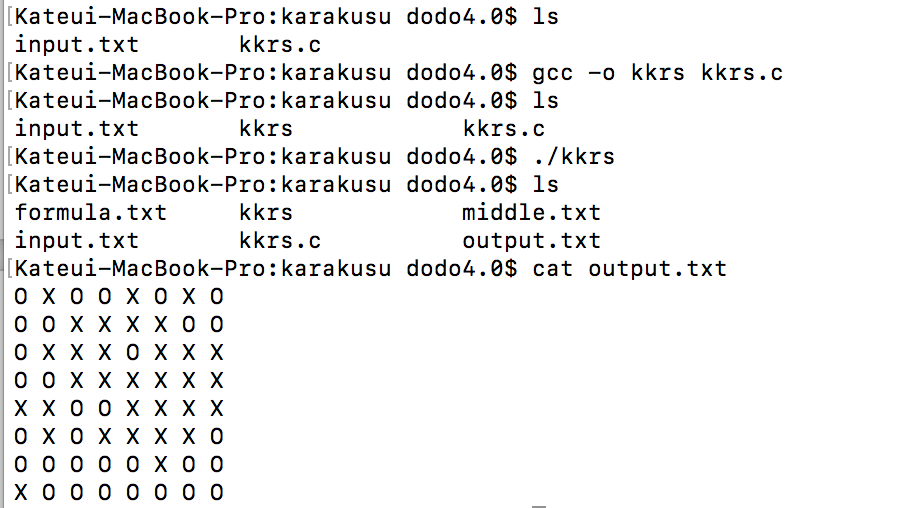
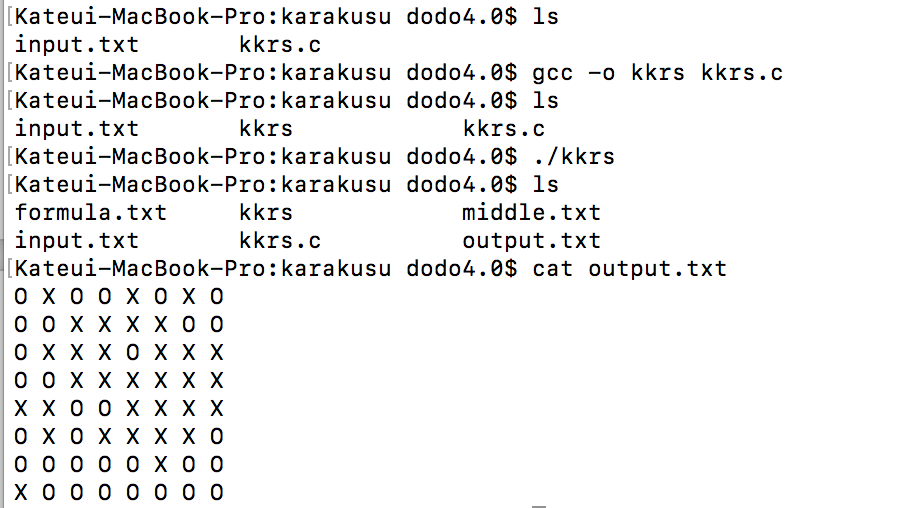


3. Run ‘kkrssol’ file.

* **./kkrssol**

4. You can see the result in ‘output.txt’

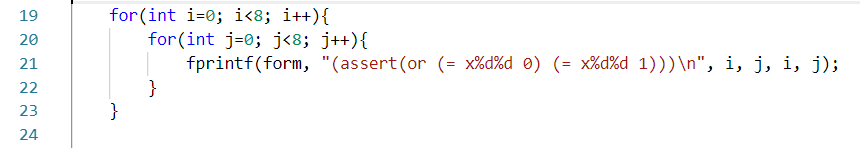
■ Demonstration that shows your program works properly



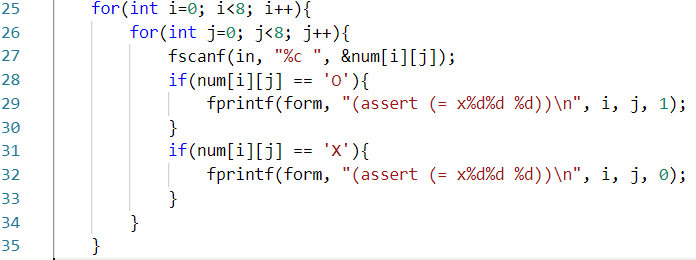
**Puzzle 3. 3-In-A-Row**

■ Description on the program designs

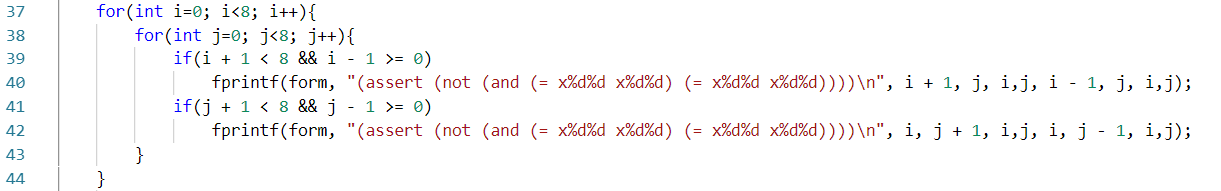
1. After the constants are declared, we make this assertion. What this assertion does is makes sure that the values of each x on the grid is either 1 or 0. We use the values 1 or 0 to signify the O or X respectively.



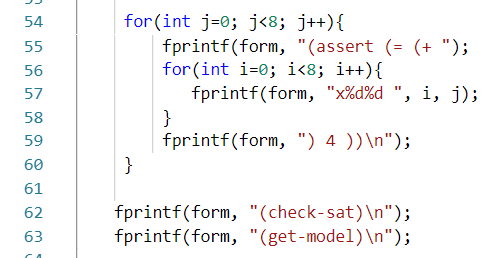
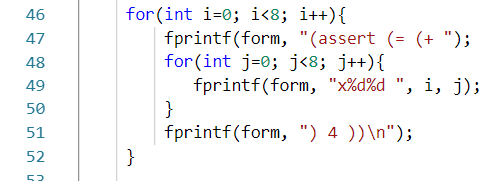
2. We use the next block of code to parse the given input to check if there are existing O’s and X’s. If there are then we match the value to the respective spot on the grid with these assertions.



3. Now that we have all the existing O’s and X’s accounted for we can start to make the assertions that help solve the puzzle. In the first if condition we check if the current spot on the grid is on the left or right edge if it is then we do not give the assertion because our assertion checks if the two spots on the grid to the left and to the right of the current spot can not both be equal to the current spot. The next if condition does the same thing but this time checks the vertical rather than the horizontal. This way there can never be three O’s or X’s in a row either vertically or horizontally.



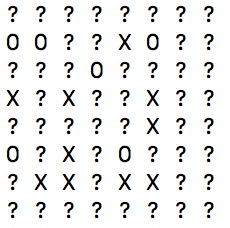
4. Finally to make sure that there are the same number of O’s and X’s in each row and each column we assert that the total of the values are equal to 4 because in an 8x8 grid for O and X to be equal to each other there must be 4 of each. Because the value of O’s and X’s are equal to 1 or 0 the total we equal to 4 if there are 4 O’s and 4 X’s.



■ Instruction on how to build and execute your programs (i.e., manual)



1. Prepare ‘input.txt’, and ‘three.c’ files. **‘three.txt’ must be in the same folder as ‘sudoku.c’**
   1. input.txt should contain the original 3-in-a-row problem in format below(8x8). Each number/symbol separated by space represents each cells in 3-in-a-row puzzle. ‘?’ represents black cells in 3-in-a-row.





2. Compile ‘three.c’ file.

* **gcc three.c -o threesol**



3. Run ‘threesol’ file.

* **./threesol**

4. You can see the result in ‘output.txt’

■ Demonstration that shows your program works properly

