CS137 Project

In this project, you will write a program to solve a KenKen puzzle.

In case you're not familiar with KenKen, here's how it works: the board is an $n \times n$ grid (with n^2 cells) where $n \geq 1$ is divided into connected regions called *cages* of various sizes. You are given a *puzzle*, which is a board with each of the cages identified as either:

- a single number (only used for cages of one cell) or
- an arithmetic expression of the form *NF*, where *N* is a positive integer and *F* is an arithmetic function (+, −, ∗ or ∕)

You are to fill in the blank cells with integers from 1 to *n* so that each number appears exactly once in every row, once in every column, and the arithmetic expression is satisfied in every cage.

We say that an arithmetic expression *NF* is satisfied if the values in the cage when combined using the function *F* return the number *N*. For example, if a cage of three elements had the requirement of 7+, the cage could (hypothetically) be filled with 1,2,4 or 1,1,5 or 1,3,3 or 2,2,3.

Note that for the operations ∕ and −, only cages of exactly two cells are allowed and that either of the two orderings of operands gives the desired number in the cage: for example, if we have 4∕, this may be satisfied by 1,4 or 4,1.

Consider the following KenKen (from www.kenken.com):

Initial puzzle

6x	3-		3
	5+	3-	
3-		2/	
	4	1-	

Solved Puzzle

6x 2	3- 1	4	3 3
3	5+ 2	3- 1	4
3- 4	3	2/ 2	1
1	4 4	1- 3	2

For example, if we are given the puzzle on the left, we would like to fill it in to form the completed puzzle on the right. Note how each number from 1 to 4 appears exactly once in each row, column, and that each cage is satisfied. We will represent the initial puzzle in C as:

```
typedef struct {
      unsigned int boardSize; // Natural integer
      unsigned int constraintsLen; // Natural integer
      entry **board; // array of (array of (entry))
      constraint *constraints; // array of (constraint)
} puzzle;
```

Where an `entry` is defined as:

```
typedef struct {
      char letter;
      unsigned int number; // Natural integer
      bool guess; // bool, if true then a guess g exists
      guess g;
} entry;
```

A constraint is defined as:

```
typedef struct {
      char letter;
      unsigned int number; // Natural integer
      char symbol; // any of: + - = * /
} constraint;
```

And, a guess is defined as:

```
typedef struct {
      char letter;
      unsigned int number;
} guess;
```

Every `entry` in the board contains each of these four types:

- a char (as shown above in the initial configuration) which indicates that the particular cell is still unknown;
- a number which indicates that the particular cell along with all other cells of the same char have been filled in;
- a bool, which when `true` indicates that the current cell has a potential answer that has not been verified in terms of the other values in the same cage. It will be `false` otherwise;
- a `guess`, which contains both the char and a possible number. **Note that all *guesses* on the board will all have the same char (i.e., they are all for the same cage). Moreover, there cannot be two *guesses* with different chars on a board.**

Each unique letter in an `entry` will also correspond to a `constraint`, which contains:

- a char letter, which is the same as the letter it belongs to;
- a char symbol, which is the arithmetic operator applied to all entries with that letter;
- a number, which is what the result must be after the operator has been applied.

To give a visual example of how an unfinished puzzle struct looks:

```
puzzle {4, 9,
     {{'a', 'b', 'b', 'c'},
      {'a', 'd', 'e', 'e'},
      {'f', 'd', 'g', 'g'},
      {'f', 'h', 'i', 'i'}},
     {{'a', 6, '*'},
      {'b', 3, '-'},
      {'c', 3, '='},
      {'d', 5, '+'},
      {'e', 3, '-'},
      {'f', 3, '-'},
      {'g', 2, '/'},
      {'h', 4, '='},
      {'i', 1, '-'}}
}
```

Where 4 is the `boardSize`, 9 is the `constraintsLen`, and the last two are the `board` and `constraints`, respectively.

Notice that the basic configuration is *n* lists of size *n* (in this case *n* = 4) and each cage is represented by a unique char, with a secondary association struct indicating what the specific cage should evaluate to. Also notice that to keep the format of the operations consistent, we use the char `'='` to represent that the cage (of size one) has no operation.

Finally, two more structs have been included to help with solving the assignment:

```
typedef struct {
      int *arr; // array of (int)
      unsigned int len; // Nat
} numArr;

typedef struct {
      puzzle *arr; //array of (puzzle)
      unsigned int len; //Nat
} puzArr;
```

These structs allow us to create arrays of both puzzles and ints, and easily keep track of the length as well.

The provided file `kenken_start.c` contains examples highlighting the various entries that a puzzle board can contain.

The file `kenken_start.c` also contains the outline of a program to solve KenKen puzzles. Your task is to complete the definitions of the functions; most parts of the documentation are provided as well with partial tests. Follow the instructions below, and make sure the given `assert` work. Be sure to add your own tests! But don't try big puzzles.

This assignment will solely be marked based on correctness tests and style.

Solving puzzles like KenKen involves searching a graph. The graph searching algorithm (*solve_kenken*) is given to you, but you need to provide two functions: a function to determine if a puzzle is complete, and a function to return a list of legal "next moves" if it is not complete. You will be solving the KenKen puzzle one constraint at a time, which may or may not be in consecutive rows or columns.

I have broken these two functions into several steps to help you along. I wrote the recursive key algorithm for you, however, I implemented not the most efficient solution but the easiest for you to complete, thus don't try it on large puzzles than the provided ones as it will take a very very long time. (Maybe after the term is over you want to continue improving this game 😊)

Note that we will test each of these functions separately, so you can get quite a few part marks on this program by completing some of the functions, even if your whole program is not complete.

For this assignment, we defined several structures for you. Do not change or add anything to those structures' definitions.

I also defined several functions for you to read/print puzzles from/to files, as well as several functions to help you in testing. I also created functions to help you copying puzzles in deep copy to avoid aliasing and failing to complete this assignment 😊. (You are welcome!). Do NOT change them. Don't spend too much time learning how they were implemented, just learn how to use them.

I am also providing several input-text files and expected output results to compare your results with. Make sure you save a copy of those files in a different folder to compare your printed files with our output files.

After you finish this assignment and submit it successfully, you are free to continue learning the parts we provided for you or even create your own version.

More tips:

Test each function thoroughly on simple cases

use valgrind to test memory leak, however make this the last thing to check as only a few tests will check for memory leak. So work on correctness first.

If you passed the provided tests for each function, probably your solution is very close to be correct.

If you did not pass the provided tests, try testing each function individually with more tests for different cases.

If you want to create more files for testing, make sure to follow the following constraints:

1) There is no space after the end of each line

2) there is only one space between each two data items on the same line.

Once you reviewed all that was provided for you and you learned and understood all that is provided in this document up to this point, you are ready to move to the next page and start solving this assignment.

Advice, solve the parts in order, You may define additional functions, you may use any function defined in previous parts, thus solve this question starting from part a, then move to b, then to c, etc.

You will notice that in the main function the second last line is:

```
// assert(testing_solve_kenken_visual());
```

commented out that you can use it any time you want to see a visual solution printed out on the screen (including the steps of solving a puzzle). This function is similar to testing_solve_kenken that prints the solution in files. This is extra to make solving this puzzle more fun. It has two purposes, the first one is to help you visualize the steps of solving a puzzle, and the second one is that it might help debug if you pay attention to the steps.

All the tests in this function are commented out so you can decide which puzzle you want to print the steps on the screen. It looks best on CS student environment. It puzzle looks best when running gcc in your student account

Have fun!

Don't move to next page unless you fully understand the first 7 pages of this document and reviewed kenken_start.c

Do NOT collaborate with others. You are not allowed to discuss this assessment with anyone at all.

You may only email the instructor for clarification questions (not debugging questions) or post privately on Piazza.

(a) Write the function `find_blank` which takes a `*puzzle` and returns a position (of type `posn`), depending on the outcome:

- the position (of type `posn`) of the first cell we wish to fill in;
- `{-2, -2}` if there are no more cells to fill in (i.e., the puzzle is complete);
- `{-1, -1}` if the string in the first constraint has only *guess*es on the board.

By "first cell to fill in", we mean the cell containing the first char in the *constraints* of the given puzzle; if there is more than one occurrence of the first constrained char, pick the topmost one, and if there is more than one occurrence of the first constrained char in the same row, then the leftmost one of those is the first. There are two ways to determine if all cells are filled in:

- all cells on the board will contain numbers, or
- the list of constraints will be empty.

The above two conditions will occur simultaneously for all valid puzzles. The location of the first cell to fill in should be returned as a `posn`. The top left corner of the puzzle is `posn{x=0,y=0}`, the top right is `posn{x=n-1,y=0}`, and the bottom right is `posn{x=n-1,y=n-1}` where `n` is the size of the puzzle.

(b) Write the functions `used_in_row` and `used_in_col`, both of which take a `*puzzle` and a position (`posn`) and returns a `numArr` struct, containing an array of the numbers used (either as *guess*es or as placed numbers) in the row or column (respectively) of the given position, and the length of that array. The returned `numArr` array should be in increasing order. Note that we will never give you a puzzle that already has two of the same number in the same row or column.

For both parts b) and c), you can assume every number used fits within the bounds of the puzzle size.

Hint: you might find the function `qsort` useful.

(c) Write the function `available_vals` which takes a `*puzzle` and a *position* and returns a `numArr` struct, containing an array of the numbers that could be placed in this cell and the length of that array. Ignore any arithmetic constraints imposed by the cage.

This function will use `used_in_row` and `used_in_col` to construct a list of all legal numbers to put in the given position of the given puzzle.

The returned `numArr` array should be in increasing order (that is, `{1, 3, 4, 6}`, not `{3, 6, 4, 1}`) and should have no duplicate elements.

(d) Write the function `place_guess` which takes a `*puzzle`, a position to fill in (as a `posn`), and value to place inside the *guess* structure. The function `place_guess` should return the board (listof (listof entry)) with the guess (`bool`) turned to `true` at the given position, and its `guess` structure filled in. You can assume the given does not already contain a number or guess. This function is called by `fill_in_guess`, which has been provided for you:

do not modify `fill_in_guess`.

(e) Write the function `guess_valid` which takes a `*puzzle` and returns `true` if the *guess*es on the board of the given puzzle satisfy the (first) *constraint* of the given puzzle, and `false` otherwise. That is, the function should return `true` if the values of all the guesses, combined using the arithmetic operator satisfy the constraint.

Be sure to test carefully for division and subtraction.

This function will only be called when all occurrences of the string of the first constraint are guesses on the board.

Tip: You may want to consider to define reduce for addition and multiplication

(f) Write the function `apply_guess`, which takes a `*puzzle` and returns a `puzzle` where all of the *guess*es have been converted into their corresponding natural number and the first constraint in the list of constraints has been removed. **Thus, you can assume that all the guesses are valid**.

(g) Write the function `neighbours` which takes a `*puzzle` and returns `puzArr` (This is the key function that will be used by `solve_kenken`). It takes a `*puzzle`, and returns an array of the valid next puzzles obtainable by either filling in the "first" cell for the string defined in the first constraint with a guess, or by applying each guess in the puzzle, so long as the guesses are all valid. It also returns the length of the array. If the guess is not valid, the guess is not applied and no neighbour is generated from that guess. Therefore, if no neighbours exist the length of puzzles will be zero. The neighbours in the returned list should be increasing order of the guess value.