Hitori

Assignment 2: Grid Parser

The objective of this assignment is to develop a grid parser that reads grids from files and writes grids to files. Grids are square arrays of 3 to 15 elements per row/column, each element is between '1' and the grid size ('1' to '3' for a 3x3 grid, '1' to 'f' for a 15x15 grid). For now, the grids are not solved, and the notion of color of each cell is not relevant yet. In this assignment only the syntax of grids is important, regardless of their solvability.

```
4 4 3 5 5
1 4 2 1 4
4 2 5 3 1
4 5 3 1 5
5 1 4 2 3
```

```
3 1 7 3 4 8 8 5
2 6 4 6 5 6 1 3
7 3 7 4 7 6 8 6
4 5 6 6 8 2 8 1
5 8 7 5 2 6 3 6
1 5 3 6 6 7 4 8
3 4 2 1 2 5 2 7
8 2 1 6 3 7 5 2
```

The file format is as follows (see examples above):

- Comment lines: all characters in a line that follow a '#' are ignored (until '\n' or EOF)
- A row has a number of significant characters ranging from 3 to 15 depending on the grid size, and end with an '\n' or EOF
- The number of rows and the number of characters per row are always the same (square grids)
- Characters for each cell are between 1 and the size of the row/column (chars 'a' to 'f' are used for numbers from 10 to 15).
- Supported separators between significant characters are tabs ('\t') and spaces (' '). Any (non-zero) number of separators is allowed, as well as lines with separators only.

If one of these rules are not satisfied the program must write an error message on `stderr`, explaining which rule has been broken, at which line, and return EXIT_FAILURE.

1. Utility functions to manage grids

1. Create two new files `src/grid.c` and `include/grid.h` in the project hierarchy and modify the Makefile in order to compile it as a module. Use `#ifdef/#define/#endif GRID_H` as in the first assignment to manage multiple inclusions of the file. These files will contain all basic utility functions for managing grids.

The data structure used to handle grids will be the following:

```c
typedef struct {
  int size;         // Number of elements in a row
  char **grid;      // Pointer to the grid
} grid_t;
```

Note: the data structure assumes the grid will be allocated as an array of cells (of `size*size` bytes). It is also allowed that you declare the grid as `char **grid`, with grid an array of pointers on rows, each row of `size` bytes.

2. Write the following functions
    a. Grid allocation (and initialization to empty cells):
    `void grid_allocate(grid_t *g, int size);`
    b. Grid deallocation:
    `void grid_free (const grid_t *g);`
    In case memory allocation fails, the program will stop and return EXIT_FAILURE.
    Suggested functions: `malloc(), calloc(), free()`
3. Write a function that prints the content of a grid in text format (see start of this sheet) in an opened file:
    `void grid_print(const grid_t *g, FILE *fd);`
    Suggested functions: `fputs(), fprintf()`
    In order to facilitate automated testing in the future, the output will not contain any comment line, will have exactly one line per row and will use only spaces as separators. For now the colors of the cells can be ignored.

2. Grid parser

The parser, that will be located in source file `src/grid_io.c`, must be able to read a grid with only one scan of the file. The scanner has to be as robust as possible when a user provides an incorrect grid file. A meaningful error message must be issued in these situations, all dynamically allocated memory has to be freed and the program has to stop and return `EXIT_FAILURE`. The parser will stop at the first error found, without any attempt to recover from the error.

1. Write a function that checks if a character is in the list of significant characters: '1', '2', '3' to 'F'. The function should return `true` if the character is in the list, `false` otherwise. The function signature is as follows:
    `bool check_char(const char c);`
2. Write the parser such that it starts with the first row and stores all the significant characters in a local array: `char line[MAX_GRID_SIZE];` When a newline character is read, the grid size is revealed and function `grid_allocate()` can be called. Then, the grid contents can be filled-in. The function prototype will be:
    `void file_parser(grid_t *grid,FILE *grid_file);`
    Suggested functions: `fgetc();`
    Display the parsed grid using function `grid_print()` to help testing your code.
3. Add to your parser the possibility to ignore comment lines (starting by character '#')
4. Modify your code to detect the following problems:
    a. Characters in the grid are not among authorized ones
    b. A row does not have the correct number of items
    c. A grid does not have the right number of columns

Error messages must be as follows:

```
sh> ./hitori grid01.txt
takuzu: error: 'grid01.txt': wrong character 'X' at line 6
sh> ./ hitori grid02.txt
takuzu: error: 'grid02.txt': line 7 is malformed (wrong number of columns)
```

For every error, write the error message on `stderr`, mention the line number in the input file when applicable, free all allocated memory and quit the program with the EXIT_FAILURE error code.