Sudoku

Introduction

It's stricly forbidden to use the `@` operator, the `List` module or any others.

Sudoku

> *"Sudoku, originally called Number Place, is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which typically has a unique solution."*
> Quote : Wikipedia.

During this pratical, we are going to ask you to write an algorithm which will help us to solve Sudoku puzzles (the easy ones, no ambiguity during the solving).

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

References:

```
# let null = 0
  and values = [1;2;3;4;5;6;7;8;9]
  and grid_sample =
    [[5; 3; 0; 0; 7; 0; 0; 0; 0];
     [6; 0; 0; 1; 9; 5; 0; 0; 0];
     [0; 9; 8; 0; 0; 0; 0; 6; 0];
     [8; 0; 0; 0; 6; 0; 0; 0; 3];
     [4; 0; 0; 8; 0; 3; 0; 0; 1];
     [7; 0; 0; 0; 2; 0; 0; 0; 6];
     [0; 6; 0; 0; 0; 0; 2; 8; 0];
     [0; 0; 0; 4; 1; 9; 0; 0; 5];
     [0; 0; 0; 0; 8; 0; 0; 7; 9]];;
```

Level 0: Must-do

0.1 Length

Write the `length` function that calculates the number of items in a list.

```
 val length : 'a list -> int = <fun>
```

0.2 Flatten

Write the `flatten` function that transforms a list of lists into a list.

```
 val flatten : 'a list list -> 'a list = <fun>
# flatten [[1;2;3];[4;5]];;
 - : int list = [1; 2; 3; 4; 5]
```

0.3 Check

Write the `check` function that indicates if an element of the list matches (using the function passed as the first argument) the one given in parameter.

```
 val check : ('a -> 'b -> bool) -> 'b -> 'a list -> bool = <fun>
```

0.4 Remove

Write the `remove` function that removes the first occurrence of an element in the list using the function given in parameter.

```
 val remove : ('a -> 'b -> bool) -> 'b -> 'a list -> 'a list = <fun>
```

0.5 Uniqueness

Write the `list_uniq` function that will remove the duplicate items in the list (found by using the function) and will output the filtered list. The element order is not important.

```
 val list_uniq : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
# list_uniq (=) [1;2;3;5;6;8;0;0;0;2;0;3;0;5;6;0;1;4];;
 - : int list = [8; 2; 3; 5; 6; 0; 1; 4]
```

0.6 Uniqueness - again

Write the `list_match` function that will return all the elements present in the two lists (but only one occurrence of each, found by using the function). The element order is not important.

```
 val list_match : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list = <fun>
# list_match (=) [1;2;3;5;6;8;0;0;0] [2;0;3;0;5;6;0;1;4];;
 - : int list = [8; 2; 3; 5; 6; 0; 1; 4]
```

1 Level 1: Matrices

1.1 Rectangle

Write the `grid_make_rectangle` function that returns a list containing y lists containing x null values (specified in parameters).

```
val grid_make_rectangle : int -> int -> 'a -> 'a list list = <fun>
```

1.2 Square

Write the `grid_make_square` function that returns a list containing x lists containing x null values (specified in parameters.)

```
val grid_make_square : int -> 'a -> 'a list list = <fun>
```

1.3 Grid

Write the `grid_make` function that returns a list of lists, which can hold the Sudoku values (specified in parameters) filled with null values (also specified).

```
val grid_make : 'a list -> 'b -> 'b list list = <fun>
```

2 Level 2: Extractions

2.1 Row

Write the `extract_row` function that returns the line number `n`. The lines are numbered from `0` to the number of allowed values - 1 (`length values - 1`), from top to bottom. The function must return a simple list.

```
val extract_row : 'a list list -> int -> 'b list -> 'a list = <fun>

# extract_row grid_sample 5 values;;
 - : int list = [7; 0; 0; 0; 2; 0; 0; 0; 6]
```

2.2 Column

Write the `extract_column` function that returns the column number `n`. The columns are numbered from `0` to the number of allowed values - 1 (`length values - 1`), from left to right. The function must return a simple list.

```
val extract_column : 'a list list -> int -> 'b list -> 'a list = <fun>

# extract_column grid_sample 6 values;;
 - : int list = [0; 0; 0; 0; 0; 0; 2; 0; 0]
```

2.3 Square

Write the `extract_square` function that returns the sub-grid number `n`. The sub-grids are numbered from `0` to the number of allowed values - 1 (`length values - 1`), from left to right, and from top to bottom. The function must return a simple list.

```
val extract_square : 'a list list -> int -> 'b list -> 'a list = <fun>

# extract_square grid_sample 8 values;;
 - : int list = [2; 8; 0; 0; 0; 5; 0; 7; 9]
```

2.4 Display

Write the `grid_print` function that displays the grid on the standard output. The function takes the element display function as a parameter.

```
val grid_print : ('a -> 'b) -> 'a list list -> unit = <fun>

# grid_print (print_int) grid_sample;;
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
- : unit = ()
```

3 Level 3: Checks

3.1 Exist and Uniqeness

Write the `list_validate` function that allows us to check if our allowed values are used only one time (maximum) in the other list. We will use the provided compare function. Be carreful with the null value.

```
val list_validate : ('a -> 'b -> bool) -> 'a list -> 'a -> 'b list -> bool = <fun>

# list_validate (=) values null [1;2;3;4;5;6;7;8;9];;
- : bool = true

# list_validate (=) values null [0;1;3;4;0;0;0;1;9];;
- : bool = false

# list_validate (=) values null [1;3;5;0;4;6;0;0;2];;
- : bool = true
```

3.2 Valid grid

Write the `grid_validate` function that checks if all the values in the grid are well placed according to the Sudoku rules (never twice the same value in a row, column or sub-grid). Be careful with the null value.

```
val grid_validate : ('a -> 'b -> bool) -> 'b list list -> 'a list -> 'a -> bool = <fun>

# grid_validate (=) grid_sample values null;;
- : bool = true
```

3.3 Full grid

Write the `grid_isfull` function that checks whether a grid is full or not, meaning whether it contains no null values.

```
val grid_isfull : ('a -> 'b -> bool) -> 'b list list -> 'a -> bool = <fun>

# grid_isfull (=) grid_sample null;;
- : bool = false
```

4 Level 4: Solving

4.1 Missing value

Write the `find_missing` function that returns the list of values, first list, un-used in the second list.

```
 val find_missing : ('a -> 'b -> bool) -> 'b list -> 'a list -> 'b list = <fun>

# find_missing (=) values [8;2;3;5;6;0;1;4];;
 - int list = [7; 9]
```

4.2 Find

Write the `grid_find` function that lists all the possible values for an empty grid's cell.

```
 val grid_find : ('a -> 'a -> bool) -> 'a list list -> int -> int -> 'a list -> 'a list = <fun>

# grid_find (=) grid_sample 8 0 values;;
  - : int list = [2; 4; 8]

# grid_find (=) grid_sample 0 8 values;;
  - : int list = [1; 2; 3]

# grid_find (=) grid_sample 0 6 values;;
  - : int list = [1; 3; 9]
```

4.3 Next-step solving

Write the `grid_nsolve` function that gives us the next grid in the solving process. We intend to iterate over the whole grid, and for each value:

- if the value is not null, we keep it
- if the value is null, we will check all the possible values to replace it
  - if we have more than one possibility, we skip the solving for this iteration on this value, and keep the null value.
  - if we have only one possibility, we substitute the null value with this possibility for the new grid.

At the end we return the new grid as a result.

```
 val grid_nsolve : ('a -> 'a -> bool) -> 'a list list -> 'a -> 'a list -> 'a list list = <fun>

# grid_print (print_int) (grid_nsolve (=) grid_sample null values)
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 5 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 7 2 8 4
0 0 0 4 1 9 0 3 5
0 0 0 0 8 0 0 7 9
 - : unit = ()
```

5 Level 5: Bonus - Almost mandatory :)

5.1 Full solving

Write the `solve` function that solves a Sudoku's grid.

```
val solve : 'a list list -> ('a -> 'a -> bool) -> 'a list -> 'a -> 'a list list = <fun>

# grid_print (print_int) (solve grid_sample (=) values null;;
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
- : unit = ()
```

6 Level 6: Bonus

6.1 True solving

For the mandatory part of this subject we deal only with grids having no ambiguity on the value during the solving process. But this is only a small part of the iceberg. We want to manage choices during solving, some of which may lead us to invalidates grids.

6.2 Grid generation

Now that we can solve any grid we find (if possible), we want to create our own. We must start from a full valid grid, and substitute some of the values by the null values. The grid complexity will depend on where and how many values we will substitute.

7 Level 7: Bonus *bored, so what's next?*

Because you have found this practical so easy (we do agree on this) we are bored to death, so we want to do something more entertaining. I'm inviting you to visit the wikipedia webpage and implement other kinds of Sudoku, such as:

- Nonomino
- Killer Sudoku
- Hyper Sudoku
- ...