EDAF95: Lab 4
The Sudoku solver, ver. 2.2

Adrian Roth and Jacek Malec

April 23, 2025

The goal of this lab is to understand how a complete Sudoku puzzle solver can be implemented in HASKELL. In the process you will learn about *monads* and the *bind* function.
All functions you are asked to write during the labs are meant to be used as inspiration for solving the assignments. If you find an approach that you think is more logical and easier to understand, we would be happy to hear about it!

To pass this lab the preparation tasks should be completed prior to the lab and then all tasks should be completed and presented to the teaching assistant.

Preparatory exercises: Reusing old concepts

Your previous struggles with the first assignment will now pay off since you will reuse the concepts and benefit from your understanding of peers and squares. You should be able to reuse most of your code from earlier labs, however, we will need to adapt a couple of types for our current purpose.

We will use the following definitions:

```
type BoardProb = [(String, [Int])]
allDigits :: [Int]
allDigits = [1, 2, 3, 4, 5, 6, 7, 8, 9]
infAllDigits = repeat allDigits
emptyBoard = zip squares infAllDigits
```

Note that our `BoardProb` object contains now all *potential* values for a square, instead of just one single value, as we had it during earlier labs!

```
parseSquare :: (String, Char) -> BoardProb -> Maybe BoardProb
parseSquare (s, x) values
  | x == '.' || x == '0' = return values
  | isDigit x = assign (digitToInt x) s values
  | otherwise = fail "not a valid grid"

parseBoard :: String -> Maybe BoardProb
parseBoard = foldr ((=<<) . parseSquare) (Just emptyBoard) . zip squares
```

The `parseBoard` function is not yet working since the `assign` function is not implemented, so you will have to think more on this problem during the lab.

First discuss with your peer what is the `BoardProb` type and how is the `infAllDigits` used?

Now implement a bunch of functions which will appear useful later. If the type declaration is not sufficiently clear, discuss it with your peers (e.g. on discord or during the lab).

Task 1: Implement the function:
`map2 :: (a -> c, b -> d) -> (a, b) -> (c, d).`
Task 2: Implement the function:
`mapIf :: (a -> a) -> (a -> Bool) -> [a] -> [a].`
Task 3: Implement the function:
`maybeOr :: Maybe a -> Maybe a -> Maybe a.`

Task 4: Implement the function:
`firstJust :: [Maybe a] -> Maybe a.`
Task 5: Implement the function:
`lookupList :: Eq a => a -> [(a, [b])] -> [b].`

These are helper functions that will take care of possible (partial) solutions of a Sudoku puzzle.

Part 1: Binding your knowledge with bind

The use of the bind function will be a required part of the solver in Assignment 2F and therefore we will cover it more thoroughly in this part of the lab. We will use the bind function on a `Maybe` data type and therefore we can write a specific type signature for it:
`maybeBind :: Maybe a -> (a -> Maybe b) -> Maybe b`
This function takes a `Maybe a` data type object as its first parameter, while the second argument is a function. This function takes a (pure) type `a` and after manipulation and possible conversion returns the type `b` inside the `Maybe` context. You should remember from the last assignment that a `Maybe` type is often used when there is a possibility of failure. We want to use `Maybe` in the same manner, where the failure is an unsolvable Sudoku and we can phrase our usage of the bind function as: We want to apply a function that might fail on a value that, in turn, might already be the result of an earlier failure.

Let us get through an example. We want to replace multiple items in a list and we also only want the result to be valid if each replacement replaces a valid element.

Task 1: Implement the function:
`maybeBind :: Maybe a -> (a -> Maybe b) -> Maybe b`
Hint: If the first argument is `Nothing` the function should return `Nothing`. Oth-

erwise it should return the result of the second parameter function applied on the value inside the first parameters `Maybe` context.

Task 2: Copy the function:

```
tryReplace :: Eq a => a -> a -> [a] -> Maybe [a]
tryReplace _ _ [] = Nothing
tryReplace y y' (x:xs)
  | x == y = Just (y':xs)
  | otherwise = fmap (x:) $ tryReplace y y' xs
```

How does it work? What does the `fmap` function do?

Now we want to successively replace the first occurrence of different items in a list. Firstly let us say we have a list:
`[1,2,3]`
We want to, in the following order, replace:

- 1 with 3
- 3 with 2

This should in the end return a `Just [2,2,3]`.
A second sequence:

- 1 with 3
- 1 with 2
- 2 with 1

should instead return `Nothing` since the second replacement attempt is invalid. To implement the first replacement sequence we can use `case` statements and get:

```
doIt :: Maybe [Int]
doIt = case tryReplace 1 3 [1,2,3] of
  Nothing -> Nothing
  Just list1 -> case tryReplace 3 2 list1 of
    Nothing -> Nothing
    Just list2 -> tryReplace 2 1 list2
```

However, this is neither pretty nor generic for use with larger sequences. As you might have guessed we can instead use our bind function (in infix) to get:

```
doIt :: Maybe [Int]
doIt = Just [1,2,3] ‘maybeBind‘ tryReplace 1 3  ‘maybeBind‘
     tryReplace 3 2 ‘maybeBind‘ tryReplace 2 1
```

This implementation can be much easier expanded into recursion and folding. This is why the bind function is so useful.

Task 3: Write a function
`recursiveReplacement ::  (Eq a) => [a] -> [a] -> [a] -> Maybe [a]`
which can perform the task explained above through recursion with `maybeBind` and test it with 2 or more sequences of your own.

When you are familiar with the `maybeBind` function you can now, if you want, use the real bind function instead:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

This is a generalised version which works for all data types that are instances of the type class `Monad`. Conveniently, `Maybe` is an instance of the `Monad` type class which means you can use it for the assignment. It might be useful to be aware of the flipped version of the bind function as well:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
```

Part 2: The `assign` function

The goal of this part is to implement a function that assigns a value to a square and then propagates elimination of that value from all the peers of the square. For each elimination of a value from a square, where we might run into a Sudoku which is unsolvable. We will consider three different special cases:

- After elimination there are no more possible values to insert in this square.
- The elimination does nothing since the value has already been eliminated from this square.
- There is only a single possible value left to put in this square.

These cases will be taken care of in a help function to the `assign` function, called `eliminate`. But first you will implement two other help functions.

Task 1: Implement the functions:
`setValue, eliminateValue ::  Int -> String -> BoardProb -> BoardProb`.
The first function sets an `Int` value to the string square in the `BoardProb` and the `eliminateValue` does the same for elimination.
Hint: Use the functions `mapIf` and `map2` you have prevously implemented.

Task 2: Implement the function:

`eliminate :: Int -> String -> BoardProb -> Maybe BoardProb`
which considers the special cases mentioned above to try and eliminate a value from a square. If unsuccessful it should return the `Maybe` value for failure.
Hint: Use guards and the `where` syntax and some of the previously implemented functions.

Task 3: Implement the function:
`assign :: Int -> String -> BoardProb -> Maybe BoardProb`.
Hint: Start by using a help function `assign'` for recursion of the peers in the elimination of a value together with the bind function.

Task 4: Now try to parse a Sudoku with the `parseBoard` function.

We will now look at how you can find a solution to a given `Board`.

Part 3: Solving the puzzle

The remaining board, after parsing, will be solved simply by trying every possible combination and the first one that works, if it exists, will be used as the solution. To begin with, we can write a help function
`solveSudoku' :: [String] -> BoardProb -> Maybe BoardProb`.
This function takes a list of squares as its first argument, which it can go through recursively. For each recursive step it will try to assign one of the values which are still possible to insert into the square (as found in the board argument). If assigning the value is successful then continue to the next square, one step deeper in recursion. Then if none of the possible values to put in the square are valid (according to the `assign` function) return up one step and continue with the next possible value for the square in this recursion step.
This might sound tricky but if bind and `map` are used together with the implemented functions `assign`, `firstJust`, and `lookupList`, you can write one beautiful recursive expression where the lazy evaluation of HASKELL will do the rest.

Finally, use this function together with the `parseBoard` to write the function `solveSudoku :: String -> Maybe BoardProb` and test that it works as expected.