EDAF95: Lab 3
Working with IO, ver. 3.2

Adrian Roth, Karl-Oskar Rikås, Jacek Malec

April 10, 2025

The goal of this lab is to use your Sudoku board implementation in a program performing I/O so that you can automate some functionality, in particular related to testing, and figure out how to make your own interaction loop.

Preparatory exercises: IO annotation

The concept of `IO` type class in Haskell is quite unlike anything one can encounter in many other programming languages. Firstly, the `IO` type class is used to distinguish *pure* computations from computations where side effects must be considered. A value of type `IO a` is like any other first-class value that you can pass around, however `IO` indicates that this value represents some kind of side-effect. So what are side effects? Functions affected by side effects are also described as *impure* as they can give different results for the same input values. Their opposite, pure functions, behave "properly" and will always return the same result for the same input. Examples of functions with side effects are functions that create random numbers, take input from files or users and write output to files or output devices.

A function with side effect should return a value in the `IO` context (more formally, in the `IO` monad), like `func :: a -> IO b`. `IO` is a monad, similarly to e.g., `Maybe`, but with one big difference:

- You can extract a value from `Maybe` with its constructor and write a function such that the output depends on the value inside:
  `f1 :: Maybe Int -> String`
- but you cannot extract a value from the `IO` context so that the output depends on the value inside it:
  `f2 :: IO Int -> String`. (NOT POSSIBLE!)

Thus `IO` is special because a value from a side-effect-influenced operation cannot escape its type.

Task 1: Write a simple function called
`giveMeANumber :: IO ()`
that reads two integer values from input, and prints a random number in between those values, using the function `randomRIO :: (a, a) -> IO a`.
Hint: Import `System.Random`, and use
`read :: Read a => String -> a`
to parse the input to the correct type.
(Should be) **Impossible challenge:** If you do think you can write this function as something not returning an IO (and, of course, not using `unsafePerformIO`), why not try.

Task 2: Choose one `IO` function that you think either will be useful in the last assignment or is interesting. Try to understand how it works and why it needs to be `IO`.
Hint: Try browsing the `IO` prelude documentation on Hackage `https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html`, focusing on the *Basic Input and Output* section.
Challenge: Find a function that the lab supervisor is not familiar with.

Task 3: Write the function `printSudoku :: [(String, Int)] -> IO ()` that takes a Sudoku board and pretty prints it to a table-like structure.
Hint: `show :: Show a => a -> String`
is useful to convert a number into a string. Besides, the `do` notation might come handy, see lecture 6.

Lab Assignment: Reading and Writing Sudoku

Task 1: Adapt your function for pretty printing Sudoku so that it can indicate where in a Sudoku something is wrong (both simple and blocked conflicts, see lab 2).

Task 2: Write a HASKELL program (with `main` function) for reading a file with multiple Sudoku in a format similar to one in `http://norvig.com/easy50.txt`. Use your `verifySudoku` function to check which of the freshly read boards are valid. You can assume one size (9x9) for all boards in the file.
Note: make sure that you can detect the format of Sudoku (dots or zeroes, bars in between) while reading it and make sure you can adapt to it. There are three test files on canvas for that purpose. **Challenge:** For the invalid ones, pretty print the board together with some indication in which unit the error was found.