AFP Assignments

(version: October 16, 2015)

Atze Dijkstra

(and Andres Löh, Doaitse Swierstra, and others)

Summer - Fall 2015

Contents

Introduction

This is the set of assignments for various AFP courses. From it are taken subsets which have to be made and submitted throughout the AFP course, or individual lab exercises can be done to train yourself in a particular topic. Beginners can start with Section 1. Smaller exercises in Section 2 are roughly categorized and partitioned by their main topic, but there is overlap in this categorization. Larger programming assignments have their place in Sections 3-6, these may touch upon multiple topics. Also, there is no ordering in the complexity or difficulty of each exercise, some exercises or sections carry a rough indication of its difficulty: `(*)` for beginner, `(**)` for intermediate, and `(***)` for advanced. On the various (AFP) courses websites you can find which of these exercises have to (or can) be made when, and, if required, how submission is done.

1 Beginners exercises

1.1 Beginners training, brief (*)

In this exercise we will guide you through the basics of writing, compiling and running a Haskell program. Some basic familiarity with using a computer and the command line is assumed.

1.1.1 Hello, world!

Use your favourite editor to create a source file containing the following program:

```
module Main where
main = putStrLn "Hello, world!"
```

You can give the file any name ending in `.hs`, such as `HelloWorld.hs`, but for the rest of the exercise we will assume that you named the file `Main.hs`.[1]

You can compile your program by invoking the following command from the command line:[2]

```
ghc --make Main.hs
```

This command will compile both the `Main` module, as well as any other modules it depends on.

Running your program will give the output:

```
Hello, world!
```

[1]Note that, like in Java and C♯, it is always a good idea to make sure the name of your file and the name of your module are identical, otherwise the compiler will complain when you try to import this module from another one. In Haskell the `Main` module—the one containing your `main` function, the function that will be invoked when you execute your compiled program—is a bit of an exceptional case in this respect, in that the module name must *always* be `Main`, but so long as you do not try import it from another module—which you generally won't—you can give it another file name. If there is no good reason to do so, however, you probably shouldn't and just name your program `Main.hs`.

[2]On the university computers you *must* open a command line using Start > Standard Applications > Computing Sciences > Haskell > Cabal!

Apart from compiling and then running a program, we can also run the program interactively from an interpreter:

```
ghci Main
```

You will now be presented with the prompt

```
*Main>
```

Type in `main` and press enter to start the program, again resulting in the output:

```
Hello, world!
```

Using the interpreter is more convenient when you are developing your program, as you can invoke any function defined in your program and pass it any arguments you desire. When you compile your program it will always be the function `main` that gets called.

1.1.2 Interaction with the outside world

Shouting a message to the outside world without bothering to listen to its response is somewhat boring. What we are really interested in is *interaction* with the outside world.

This can be achieved using the `interact` function from Haskell's standard library (also called the `Prelude`). The function `interact` is an IO action[3] that takes another function as its argument. This concept—passing a function as an argument to another function—may be unfamiliar if you have only programmed in *imperative programming languages*, such Java or C♯, before, but is one of the cornerstones of the *functional programming* paradigm, as is reflected in its name. The function `interact` is thus an example of a so-called *higher-order function* and we shall become intimately familiar with them during this course.

But what does the function `interact` do? First, it reads a string from the *standard input*—by default your keyboard, but below we shall see how we can redirect the standard input to read from a file instead. Next, it applies the function you passed as an argument to `interact` to the string it just read from the standard input, to transform it into another string. Finally, it prints the transformed string to the *standard output*—by default your screen, but below we shall see how we can redirect the standard output to write to a file instead.

Note that this does put some restrictions on the kinds of functions you can pass to `interact`: they should take exactly one string as their argument, and also return a

[3]IO stands for "input/output."

string as their result. Formally, we write that the argument of `interact` should be of the *type* `String` → `String`.

Time for an example:

```
module Main where
main = interact reverse
```

Additionally, create a file called `in.txt`, containing the text:

```
eb ot
eb ot ton ro
noitseuq eht si taht
```

Compile the program (running it from the interpreter is not going to work correctly!), and run it, while redirecting the standard input to read from the file `in.txt`. On Windows machines this can be achieved using the command:

```
Main < in.txt
```

On Mac and Linux machines you need to use the command:

```
./Main < in.txt
```

This will give the output:

```
that is the question
or not to be
to be
```

Almost, but not quite right. We reversed the complete file, instead of reversing the lines one at a time. Let us try again:

```
module Main where
main = interact (unlines . map reverse . lines)
```

Note that we will often try very hard to make out programs look very pretty on paper (including in the exercises, the lecture notes, and on exams) by replacing some of the operators used in the source code by their correct mathematical symbols. The actual text that you need to enter in your source file would be:

```
module Main where

main = interact (unlines . map reverse . lines)
```

This program makes use of the *function composition* operator `.` to glue three functions together before they are passed to `interact`. Note that, just like in mathematics, composed functions should be read from right-to-left.

The first function, `lines`, will split a string into a list of strings (denoted `[String]`). Its type is thus `String -> [String]`.

The second function, `map reverse`, will reverse all of the strings contained inside a list. Its type is thus `[String] -> [String]`. Note that this function is actually another instance of a higher-order function (in this case `map`) applied to a second function (`reverse`).

The third function, `unlines`, takes a list of strings and concatenates them together with a newline character in between.

In this instance it may be more apparent what the program does if we could compose functions from left-to-right. We can do so as follows:

```
module Main where
import Control.Arrow
main = interact (lines >>> map reverse >>> unlines)
```

Running either of the two programs will give us the desired output:

```
to be
or not to be
that is the question
```

If you want to save the results to a file instead of printing it on the screen, you can instead invoke the program with the command:

```
Main < in.txt > out.txt
```

1.1.3 The exercise

Modify the program given above to have it—instead of printing each line of the input on a separate line—print the lines with slashes in between. Thus for the input:

```
eb ot
eb ot ton ro
noitseuq eht si taht
```

the program should give as output:

```
 to be / or not to be / that is the question
```

Hint: you can use the function `intercalate` from the library `Data.List` (which you will have to `import`). See `http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-List.html#v:intercalate`.

1.2 Beginners training, extensive (*)

This introduction takes you slowly through your first steps with Haskell. Most of the exercises are easy, some may even sound too easy, but try them nevertheless. They're intended to give you more familiarity with the interpreter, and also to let you encounter typical error situations in a harmless situation, so that you can learn to deal with them.

Some other exercises are harder. If you have problems solving a particular exercise within 5 to 10 minutes, remember the number of the exercise and go on. Later, go back and try again. If you still fail, ask someone.

A few exercises are marked explicitly as being difficult. It is not necessary to solve them on a first pass, and they may take longer than 5 to 10 minutes. They are included as a challenge.

Exercise 1.2.1 (medium)**.** [Only if you are using your own machine – on the lab machines, GHC is already installed for you.] Install GHC, including the interpreter GHCi. For more information on how to do that, google for

```
Haskell Platform
```

Install the latest version of the platform that is available for your OS. If you want to test that everything works as expected, you can do "Haskell in 5 Steps":

```
http://haskell.org/haskellwiki/Haskell_in_5_steps
```

Exercise 1.2.2. Start up GHCi. Depending on your environment, this can either be done by choosing GHCi from a menu, by clicking on an icon, or by opening a command line and at the command line prompt `$`, typing

```
$ ghci
```

You're confronted with a welcome message and finally, the GHCi prompt, which will usually say `Prelude>`, but we'll abbreviate it here simply to `>`.

1.2.1 Getting started with GHCi

Exercise 1.2.3. Type

```
> :?
```

to get help. This lists all the *commands* available in the *interpreter* (i.e., in GHCi). Commands all start with a colon `:`. Commands are not Haskell code, they just tell the interpreter what to do. As you can see, there are a lot of commands, but we'll only need a few of them.

Exercise 1.2.4. Type

```
> :q
```

(or `:quit`) to leave the interpreter again. This is a useful command to know if you ever get (temporarily) tired of practicing Haskell.

Exercise 1.2.5. Start GHCi again.

```
$ ghci
```

1.2.2 Basic arithmetic

Exercise 1.2.6. Type

```
> 2
```

Generally, you can type *expressions* at the prompt. The expressions are evaluated, and the final value is printed.

Exercise 1.2.7. Type

```
> 2 + 2
```

There are binary *operators* in Haskell that are written in usual infix notation.

Exercise 1.2.8. Type

```
> (+) 2 2
```

All binary operators can also be written in prefix notation. The operator is then surrounded by parentheses. Arguments are just separated by spaces, no parentheses or commas.

Exercise 1.2.9. A special feature of the GHCi interpreter is that the last evaluated value is always available for further computation under the name `it`:

```
> it
```

Also try:

```
> it + 2
> it + 3
> it + it
```

Exercise 1.2.10. Operators have their standard priorities. In particular, `(*)` binds stronger than `(+)` or `(-)`. Type

```
> 6 * 11 - 2
```

Exercise 1.2.11. Parentheses can be used to direct the computation. Type

```
> 6 * (11 - 2)
```

and compare the result with the result from Exercise 1.2.10.

Exercise 1.2.12. Try

```
> (-) ((*) 6 11) 2
> (*) 6 ((-) 11 2)
```

The same as the two expressions before, but in prefix notation – just to show that it is possible.

Exercise 1.2.13. Type

```
> 6 * (11 - 2
```

If you forget parentheses or other make syntactic mistakes, you'll often get a *parse error*. This indicates that your expression isn't legal Haskell. You can press the "up arrow" key on your keyboard to get back the expression you typed, correct it, and try again.

Exercise 1.2.14. Try

```
> True
> "Hello"
```

Not only numbers can be evaluated. Haskell has several types, for instance Boolean values and strings.

1.2.3 Booleans

Exercise 1.2.15. Type

```
> True || False
```

There are special operators on Boolean values. The `(||)` operator is the logical "or". It returns true if at least one of the arguments is true.

Exercise 1.2.16. Try

```
> not True
> not False
```

Here, `not` is a (predefined) function that negates a Boolean value. The argument of `not` is just separated with a space, no parentheses are necessary. Although

```
> not (True)
```

would work, this syntax is not typically used by Haskell programmers.

Exercise 1.2.17. Of course, the logical "and" exists as well. Try

```
True && True
False && True
(not False || True) && (False || True)
```

What's the answer?

Exercise 1.2.18. Type

```
> true
```

Haskell is case-sensitive, i.e., it matters if you use lower- or upper-case characters. In Haskell, identifiers that start with a lower-case letter are abbreviations for expressions (i.e., predefined functions such as `not`), whereas identifiers that start with an upper-case letter are somewhat special – so-called *data constructors*, such as `True` and `False`.

If you type an identifier that hasn't been defined yet, you'll get an error message saying that the identifier is "not in scope." Scope refers to the area of a program where an identifier is known, so "not in scope" indicates that the identifier is unknown at this point.

Exercise 1.2.19. Try

```
> True || True && False
> not True && False
```

What does this tell you about the priorities of the operators `(||)` and `(&&)`? Also, what does it tell you about the priority of function application? How would you have to place parentheses in each of the two examples to get the opposite result?

1.2.4 Strings

Exercise 1.2.20. For strings, there are also operators. For instance, `(++)` concatenates two strings – try:

```
> "Hello" ++ " " ++ "world"
```

Exercise 1.2.21. The function `length` computes the length of a string – try:

```
> length "Hello"
> length "world"
> length ""
```

Exercise 1.2.22. Try also:

```
> head "Hello"
> tail "Hello"
> last "Hello"
> init "Hello"
> reverse "Hello"
> null "Hello"
```

What do these functions do? Gain confidence in your assumptions by trying more examples.

Exercise 1.2.23. If you type

```
> head ""
```

you get an *exception*. As you probably have discovered in Exercise 1.2.22, `head` tries to determine the first element of a string. On an empty string, it fails! Exceptions are different from all the errors you have seen so far, because they occur while the program is executed. In contrast, parse errors (Exercise 1.2.13) and scoping errors (Exercise 1.2.18) occur *before* the program is executed. We also say that exceptions are *dynamic* errors, whereas the others are *static* errors.

Exercise 1.2.24. Which other of the functions in Exercise 1.2.22 cause an exception when called on the empty string?

Exercise 1.2.25. Type

```
> "Hello
```

If you forget quotes around a string and in a few other situations, you'll get a "lexical error". Again, this indicates that you haven't provided a legal Haskell expression, and given the distinction made in Exercise 1.2.23, it is a static error. Lexical errors are much like a parse error, but at an even more fundamental level. In the "Grammars and Parsing" part of the course, you'll learn the difference between parse errors and lexical errors. You'll see that it is mainly a difference relevant for the implementation of the interpreter/compiler, and not so important for the programmer.

Exercise 1.2.26 (medium, recommended)**.** Type

```
> not "Hello"
```

If you try to use logical negation on a string, you get yet another sort of error. This is – once you get used to programming in Haskell – the most frequent kind of error you'll be confronted with: *a type error*. Type errors usually indicate that something is semantically wrong with your program. Type errors come in many flavours, and require a lot of practice to read and understand. Therefore it is actually good to make type errors, because it gives you practice understanding them!

Here, you'll get the following message:

```
Couldn't match expected type 'Bool' against inferred type '[Char]'
In the first argument of 'not', namely '"Hello"'
In the expression: not "Hello"
In the definition of 'it': it = not "Hello"
```

The first line tells you *what* went wrong, the other lines tell you more about *where* it went wrong.

The first line says that a `Bool` was expected where a `[Char]` was given. Now, as we'll see soon, `[Char]` just means a string. So, a Boolean was expected where a string was given. This makes sense: logical negation via `not` expects as its argument a Boolean value, but we've passed `"Hello"`, which is a string. Indeed, this is what the second line says: the argument of `not`, namely `"Hello"` is blamed. The other lines give more information about the context and are not so important (note, however that the last line mentions `it`, the identifier that's always implicitly bound to the last result in the interpreter, see Exercise 1.2.9).

1.2.5 Types

Exercise 1.2.27. Every Haskell expression has a type, and there is an interpreter command to ask for that type.

```
> :t True
True :: Bool
```

The symbol :: reads "has the type", so the answer is that `True` has the type `Bool` of Boolean values.

Exercise 1.2.28. Let's try a string next:

```
> :t "Hello"
"Hello" :: [Char]
```

This time, the answer is that `"Hello"` has the type `[Char]`. This in turn stands for a "list with elements of type `Char`," where `Char` is the type of single characters. So in Haskell, strings are just lists of single characters, and the notation `"Hello"` is actually just an abbreviation for a notation that makes the list-like character much more obvious – try:

```
> ['H', 'e', 'l', 'l', 'o']
> :t ['H', 'e', 'l', 'l', 'o']
> 'H'
> :t 'H'
```

Single characters are written in single quotes, strings are lists of characters, but can be written shorter between double quotes.

Exercise 1.2.29. Recall function `head` from Exercise 1.2.22. Try again

```
> head "Hello"
```

Does it result in a string or in a single character? Verify your result using

```
> :t head "Hello"
```

Note that you can apply `:t` to arbitrary expressions, not just values. How about the expression `tail "Hello"` – does it return a string or a single character?

Exercise 1.2.30. Functions (and operators) also have types:

```
> :t not
not :: Bool -> Bool
```

The type `Bool -> Bool` is the type of functions that expect a `Bool` as parameter and deliver a `Bool` as result. Now look again at Exercise 1.2.26 with the type error. There, we tried

```
not "Hello"
```

The function `not` expects a `Bool`, and `"Hello"` is a `[Char]`, hence the error.

1.2.6 Lists

Exercise 1.2.31 (medium)**.** Let's try one of our string functions next:

```
> :t length
length :: [a] -> Int
```

The result is somewhat surprising. The function returns an integer (i.e., an `Int`), ok. But it doesn't take a string, i.e., a `[Char]`, but instead a `[a]`? What does the `a` mean?

It means that we don't care! The `a` is a *type variable*. A type variable is a bit like a joker – we can choose any type to take `a`'s place! So `length` computes the length of *any* list – not just lists of characters, but also lists of numbers, or even lists of lists. Try to guess the answers before trying the expressions in the interpreter:

```
> length [1,2,3]
> length [[1,2],[1,2,3],[],[99]]
> length ["Hello","world"]
```

If we compute the length of a list of lists, the length of the inner lists is irrelevant.

Types like the type of `length`, which contain type variables, are called *polymorphic*, because it is like they have many different types at once.

Exercise 1.2.32. We can be even more adventurous. In Haskell, functions are just values like anything else. So, we can put functions into lists!

```
> length [length,head]
```

Here we have a list of two functions.

Exercise 1.2.33. Recall function `head` from Exercise 1.2.22, 1.2.29 and 1.2.32. Try to guess what the type of `head` is. Verify it in the interpreter.

Exercise 1.2.34. Guess and check the types of `tail`, `reverse` and `null`.

Exercise 1.2.35. Guess and check the type of `[True,False,False]`.

Exercise 1.2.36. All elements of a list must be of the same type! Let's try what happens if this is not the case:

```
> [True,"Hello"]
```

A type error again! And very similar to the one before:

```
Couldn't match expected type 'Bool' against inferred type '[Char]'
  Expected type: Bool
  Inferred type: [Char]
In the expression: "Hello"
In the expression: [True, "Hello"]
```

Again, we have provided a string where a Boolean value was expected. Again, `"Hello"` is blamed. This time, the first element of the list was a `Bool` (namely `True`), so the type checker inferred that we're writing a list of Booleans.

Exercise 1.2.37. Guess what the type of the empty list is! Think about this for at least a minute! Only then try it in the interpreter.

Exercise 1.2.38. Let us produce another type error.

```
> [[False],True]
```

Here, we get:

```
Couldn't match expected type '[Bool]' against inferred type 'Bool'
In the expression: True
In the expression: [[False], True]
In the definition of 'it': it = [[False], True]
```

Because the first element is of type `[Bool]`, and the second of type `Bool`.

Exercise 1.2.39 (difficult)**.** The alert reader might have discovered an apparent inconsistency in what we've discussed so far. I said: the elements of lists all have to be of the same type. We have seen in the previous exercises that `length` and `head` are of different types. Try again:

```
> :t length
> :t head
```

But we have successfully computed

```
> length [length, head]
```

in Exercise 1.2.32. Can you guess why?

Try the following:

```
> :t []
> :t [length]
> :t [head]
> :t [length, head]
```

Can you explain these types?

Exercise 1.2.40. Try to guess what

```
> (head [length]) "Hello"
```

does, and then verify your guess in the interpreter. Recall that functions are just ordinary values in Haskell! They can be passed around, put into data structures such as lists, and be arguments and results of other functions. Even though it might seem unusual, don't let it confuse you.

Exercise 1.2.41 (difficult)**.** Try to guess what

```
> (head [length, head]) "Hello"
```

does, and then verify your guess in the interpreter. Try to explain! What does this say about Haskell's type system?

1.2.7 Tuples

Exercise 1.2.42. Every element in a list has the same type, but lists can contain arbitrarily many elements. Haskell also provides *tuples*. Tuples have a fixed length, but elements of different types can be combined. Try the following expressions:

```
> (1,2)
> (1,"Hello")
> (True,id,[1,2])
> (1,2,3)
> (1,(2,3))
> ((1,2),3)
> [1,2,3]
```

Exercise 1.2.43. Ask for the types of the expressions from Exercise 1.2.42. Note how the tuple types reflect each of the types of the components. Note also that the four final expressions all have different types.

Exercise 1.2.44. Pairs are used quite often. Tuples with more components are less frequently. For pairs, there are projection functions. Try:

```
> fst (1,"Hello")
> snd (1,"Hello")
> fst (1,2,3)
> :t fst
> :t snd
> fst (snd (1,(2,3)))
```

Try to understand the type error and the types.

1.2.8 Currying

Exercise 1.2.45. As indicated before, operators have types, too.

```
> :t (++)
```

In Exercise 1.2.8, I have explained that operators, if written between parentheses, can be used in prefix notation. In fact, if between parentheses, they're treated just like ordinary function names. Therefore, we can also ask for the type of an operator if using parentheses. The answer is

```
(++) :: [a] -> [a] -> [a]
```

Since `(++)` is a binary operator, it takes two arguments. Functions with multiple arguments are usually written in so-called *curried* style – after Haskell B. Curry, one of the first persons to use this style and also the person after whom the language Haskell was named. There is no magic here. Intuitively, it means that the function gets its arguments one by one, rather than all at the same time. It gets a list, then another list, and then produces yet another list as its result. This sequentiality is reflected in the (prefix) syntax of function application:

```
> (++) [1,2,3] [4,5]
> (++) "don't " "panic"
```

The parameters are just separated by spaces, there are no parentheses or commas to group the parameters together.

Concatenation is polymorphic again. It concatenates two lists of the *same* element type (the same variable is used everywhere), and the result list also has that element type.

Exercise 1.2.46. Verify that concatenating two lists of different element type results in a type error. Try to understand the resulting type error message.

Exercise 1.2.47. Here are some more functions with two arguments – all of them have types in the curried style. Check their types and try to find out what they are doing by passing them type-correct parameters. Also try to pass type-incorrect parameters and try to understand the error messages – for instance, try:

```
> :t take
> take 5 "Hello world!"
> take 42 "Hello world!"
> take 0 "Hello world!"
> take (-3) "Hello world!"
> take True "Hello world!"
> take 7 'H'
```

Perform similar tests for the following functions:

```
drop
replicate
const
```

What's the difference between `replicate 7 'x'` and `replicate 7 "x"`? Did you succeed passing type-incorrect parameters to `const`?

Exercise 1.2.48. In the interpreter, it is possible to abbreviate expressions by giving them a name. The syntax for this is

`let` *identifier* = *expression*

The whole thing is called a *statement* and – provided that *expression* is type-correct, introduces *identifier* for further use. Try

```
> let hw = "Hello world!"
> :t hw
> hw
> length hw
```

Exercise 1.2.49. Define an abbreviation of your own.

Exercise 1.2.50. Recall Exercise 1.2.45, where we have introduced functions (or operators) with multiple arguments. I said that currying means that function parameters are passed one-by-one. This also means that not all parameters have to be provided at the same time. We can *partially apply* curried functions with multiple parameters:

```
> :t take
> :t take 2
> :t take 2 "Rambaldi"
> take 2 "Rambaldi"
> :t (++)
> :t (++) "Ramb"
> :t (++) "Ramb" "aldi"
> (++) "Ramb" " aldi"
> :t replicate
> :t replicate 3
> :t replicate 3 1
> :t replicate 3 'c'
> :t replicate 3 False
> replicate 3 False
```

Exercise 1.2.51. We can abbreviate useful partial applications and give them a name, for instance:

```
> let dup = replicate 2
> :t dup
> dup 'X'
> dup 0
> dup True
> dup ' '
> let indent = (++) (dup ' ')
> indent "Hello world!"
> :t indent
```

Exercise 1.2.52. Try the following expressions:

```
> take 3 "Hello"
> take False "Hello"
> take "x" "Hello"
> take 2.5 "Hello"
```

The first should succeed, the others fail. Try to explain why the others fail. Try to understand the error messages of the second and third case. The error message for the fourth expression is strange, and we'll explore this further.

1.2.9 Overloading

Exercise 1.2.53 (medium, recommended)**.** Ask for the type of `2.5`:

```
> :t 2.5
2.5 :: (Fractional t) => t
```

This might come as a surprise. You might have expected to read `Float` here, for floating point number, or `Double`, for a double-precision number. Instead, the type contains a variable `t`, which suggests it's polymorphic, like many of the functions we've seen so far. But in addition, there's a so-called *constraint* in this type – the part before the double arrow ⇒. The way to read this type is as a logical implication: for all types `t`, if `t` is `Fractional`, then `2.5` has the type `t`. So, it's a bit like a polymorphic type, but with an additional condition on the type variable – we can't choose an arbitrary type, but instead must choose a `Fractional` type. Types that have such constraints are called *overloaded*.

Exercise 1.2.54. What does `Fractional` mean? The identifier `Fractional` refers to a *type class*, that's a collection of types that share certain properties. You can ask the interpreter GHCi to give you more information about a type class and the types that belong to that type class:

```
> :i Fractional
```

The command `:i` (or `:info`) can be used with any identifier and results in information about where and how that identifier is defined. If used on a type class, you get to see the definition of the class, and information about which types belong to that class. In this case, you should see:

```
class (Num a) => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
      -- Defined in GHC.Real
instance Fractional Double  -- Defined in GHC.Float
instance Fractional Float  -- Defined in GHC.Float
```

The first lines are the class declaration, which tells us something about the functionality that types of this class are required to support. But for now, we are more interested in the members of the class, which are called *instances*. In this case, the class consists of two types: `Double` and `Float`, double- and single-precision floating point numbers, respectively.

1.2.10 Numeric types and their classes

Exercise 1.2.55. Recall

```
> :t 2.5
```

With what we've learned now, the type tells us that `2.5` can be a `Double` or a `Float`, depending on context, but not, for instance, an `Int`! Why not? Because `Int` is not an instance of the class `Fractional`! And this explains the strange type error message you get when trying

```
> take 2.5 "Hello"
```

The function `take` expects an `Int`, but is given something of type `(Fractional t) => t`, i.e., something that could be anything in the class `Fractional`. Now, `Int` is not in `Fractional`, hence the error.

Exercise 1.2.56. Type classes are ubiquitous in Haskell, and there are many type classes predefined. If you check the type of numberic literals without a decimal dot, you'll get a different constraint:

```
> :t 2
```

The class `Num` is larger than `Fractional`, and comprises all numeric types, not just `Float` and `Double`, but also `Int` and `Integer`. Both are types for integers, but `Int` is bounded (but at least 32-bit large), while `Integer` can can compute with arbitrarily large integers (but is slightly less efficient). Type

```
> :i Num
```

to get information about the instances of `Num`.

Exercise 1.2.57. Now, we can also look at the types of the numeric operators:

```
> :t (+)
> :t (*)
> :t (-)
> :t (/)
```

You see that all of them are overloaded. While the first three work for all numeric types, the division operator only works for fractional types.

Exercise 1.2.58. The fact that numbers you type in, such as `2`, are overloaded over all numeric types (including the fractional types), means that you can use division on such numbers:

```
> 1 / 2
> 3 / 8
> :t 3 / 8
```

The result of such a division can still be used as a `Float` or a `Double`, but no longer as an `Int`. This also holds for divisions that happen to produce a whole number:

```
> 4 / 2
> :t 4 / 2
```

So,

```
> take (4 / 2) "Hello"
```

should produce a type error again. Verify that.

Exercise 1.2.59. The types `Float` and `Double` are related (their only difference is their precision in positions after the decimal dot), so it makes sense that operations usually don't work on only `Float` or only `Double`, but instead are overloaded, with a constraint on `Fractional`.

Similarly, `Int` and `Integer` are related (their only difference is that `Int` is bounded and `Integer` isn't), and many operations work on both. So there's a class for these two types as well: `Integral`. Type

```
> :i Integral
```

and verify that `Integer` and `Int` are instances of `Integral`. The type class `Num` contains both the two integral types in `Integral` and the two fractional types in `Fractional`.

Exercise 1.2.60. An operation on the integral types is integer division, called `div`:

```
> :t div
> div 4 2
> :t div 4 2
> div 7 2
> div 11 3
```

In integer division, the result is always rounded down to the nearest integer.

Exercise 1.2.61. The counterpart to integer division is `mod`, which computes the remainder of integer division:

```
> :t mod
> mod 7 2
> :t mod 7 2
> mod 11 3
```

Exercise 1.2.62. Recall that infix operators can be used as prefix functions by surrounding them with parentheses. Similarly, normal functions can be used as infix operators by surrounding them with backquotes. For some Haskell functions, such as `div` and `mod`, this is common practice:

```
> 11 ‘div‘ 3
> 11 ‘mod‘ 3
```

1.2.11 Printing values

Exercise 1.2.63. Another type class is revealed by looking at the function `show`:

```
> :t show
(Show a) => a -> String
```

For all types `a` in `Show`, a value can be turned into a string. It turns out that many, many types are in `Show`. If you try

```
> :i show
```

you probably have to scroll up to be able to see all the instances in that class. Try it out with the types you already know, for instance:

```
> show 2
> show 2.5
> show True
> show ’a’
> show "Hello"
> show [1, 2, 3]
```

Exercise 1.2.64. GHCi uses a variant of `show`, called `print`, internally when printing the results of expressions you type! If a type is not an instance of type `Show`, then GHCi doesn’t know how to print a result.

Some types are not in `Show`, for example functions! If you try to evaluate a function, you get a type error:

```
> take
```

This results in:

```
No instance for (Show (Int -> [a] -> [a]))
  arising from use of ‘print’ at <interactive>:1:0-3
Possible fix:
  add an instance declaration for (Show (Int -> [a] -> [a]))
In the expression: print it
In a ’do’ expression: print it
```

The interesting part is the first line. There is no `Show` instance for the type `Int -> [a] -> [a]` – the type of `take`. In other words, GHCi doesn't know how to print a function on screen.

It is important to note that `take` itself is type correct, as

```
> :t take
```

demonstrates. The type error here only stems from the fact that GHCi tries to print the result of an expression you type in and implicitly computes `print it` (see Exercise 1.2.9) after evaluating the expression. That's why `print` is mentioned in the error message, even though you haven't typed it in.

1.2.12 Equality

Exercise 1.2.65. Yet another interesting class is the class `Eq` of types supporting an equality operation:

```
> :t (==)
> :i Eq
```

The equality operator consists of two =-characters. Again, there are very many types supporting equality. The result is always a Boolean value:

```
> 1 == 1
> 1 == 1.5
> True == False
> 'x' == 'X'
> 'X' == 'X'
> [1, 2] == [2, 1]
> [1, 2] == [1, 2]
> "Hello" == "world"
```

Exercise 1.2.66. Try to guess the result of

```
> True == (True == True)
```

and verify it in the interpreter. Can you explain what's happening?

Exercise 1.2.67. Function types are an example of types not supporting equality. The expression

```
> (++) == (++)
```

does not return `True`, but results in a type error explaining that the type of the concatenation operator `[a] -> [a] -> [a]` is not an instance of the `Eq` class.

Exercise 1.2.68. Find out the types of the following functions, and find out what they do by applying them to several arguments:

```
odd
even
gcd
sum
product
```

Exercise 1.2.69. The function `elem` checks whether a list contains a specific element. Check the type:

```
> :t elem
```

The function `elem` is often written infix as `‘elem‘`. Try to guess the answers before verifying them with the interpreter:

```
> 7‘elem‘ [2, 3, 5, 7, 11]
> 'e' ‘elem‘ "Hello"
> [] ‘elem‘ []
> [] ‘elem‘ [[]]
> [] ‘elem‘ [2, 3, 5, 7, 11]
```

Can you explain the type error?

1.2.13 Enumeration

Exercise 1.2.70. Type

```
> [1..5]
```

and see what happens. Also try the following expressions

```
> [1, 3..10]
> [10, 9..1]
> ['a'..'d']
```

Recall that strings are just lists of characters.

Exercise 1.2.71. Type

```
> [1..]
```

This will start printing all natural numbers starting with `1`. You will have to interrupt execution using `Ctrl+C`, i.e., by pressing the `Ctrl` key, holding it, then pressing `C`. You can do this whenever you want to interrupt GHCi, whether it is because the computation would not terminate or is just taking too long for your taste.

Exercise 1.2.72. As exercise 1.2.70 shows, ranges can be specified for several types. The types that allow this are in yet another type class called `Enum`. Type

```
:i Enum
```

You will see that there are several methods in class `Enum`, among them `enumFromTo`, `enumFromThenTo` and `enumFrom`.

The range notation using .. is so-called *syntactic sugar* for these functions. Haskell simplifies range expressions to applications of these functions. Verify that

```
enumFromTo 1 5
enumFromThenTo 1 3 10
enumFromThenTo 10 9 1
enumFrom 1
```

produce the same results as the range expressions above.

1.2.14 Defining new functions

Exercise 1.2.73. Let's define a new function:

```
> let inc x = 1 + x
```

This is like defining an abbreviation, but additionally, we introduce a parameter `x` that we can use on the right hand side. This function increases a numeric value by 1:

```
> :t inc
```

The compiler infers the best possible type (including the `Num` constraint) for our function – you don't have to provide type information explicitly. The new function is the same as the function defined via partial application of `(+)`:

```
> let inc' = (+) 1
> :t inc'
> inc 41
> inc' it
```

Exercise 1.2.74. Here is a function we couldn't have defined via partial application:

```
> let parens s = "(" ++ s ++ ")"
```

It surrounds a given string `s` with parentheses. Try it on a couple of strings. Also try it on the empty string.

Exercise 1.2.75. See how

```
> init (tail "Hello")
```

drops the first an the last element of a string. Define a function `prune` that drops the first and the last element of any string. Check the type of your function:

```
> :t prune
```

It should be `[a] -> [a]`. Apply it to a list of numbers and see if it works as well. What happens if you apply it to a list of less than two elements? What happens if you apply it to a list of exactly two elements?

Exercise 1.2.76. Here's a function `ralign` with two arguments:

```
> let ralign n s = replicate (n - length s) ' ' ++ s
```

Guess its type and what it does. Then verify in the interpreter. What does happen if `n` is smaller than the length of `s`?

Note that several parameters appear on the left hand side in a similar style as the function application would look: all arguments are separated by spaces, there are no parentheses or commas. The type of the function is in curried style (see Exercise 1.2.45), so it can be partially applied:

```
> let myralign = ralign 50
```

Test `myralign` on a couple of strings. What can you say about `length (ralign n s)` for arbitrary values of `n` and `s`?

Exercise 1.2.77. Can you write a function `lalign` that moves a string to the left rather than to the right?

Exercise 1.2.78 (medium)**.** Can you write a function `calign` that (approximately) centers a string rather than to move it to the left or right? Hint: You'll probably have to perform integer division via `mod` (see Exercise 1.2.61). Make sure that `length (calign n s)` is always `n`, and that the function works in all cases, whether `n` is even or odd, and whether `length s` is even and odd.

1.2.15 Anonymous functions

Exercise 1.2.79. We have already seen that in Haskell, functions are just normal values. This goes even further: in Haskell, functions do not require a name! Just like you don't have to give a name to every number or string you use, you also can write down functions without giving them a name. Such functions are called *anonymous* functions. A named function such as

```
inc n = n + 1
```

can be written as

```
\n -> n + 1
```

The backslash `\` and `->` are just syntax: to introduce an anonymous function, and to separate the parameters from the body of the function. The above is the function that "maps `n` to `n + 1`". The backslash `\` is read as "lambda" (for mathematical/historical reasons), so you can read the above function as "lambda `n` arrow `n + 1`". If you want to apply such an anonymous function, you have to put it in parentheses to delimit the function body: try

```
> (\n -> n + 1) 10
```

Exercise 1.2.80 (medium)**.** For the range expressions introduced in exercise 1.2.70, we can use an anonymous function and the `:t` command to determine that ranges are indeed a feature that is tied to the type class `Enum`. Check

```
> :t \x y -> [x..y]
```

and try to understand the type.

Exercise 1.2.81. Every function can be written as an anonymous function. For instance, if for some reason we wouldn't want to give `ralign` a name, we could use it anonymously:

```
> (\n s -> replicate (n - length s) ' ' ++ s) 10 "Hello"
```

Exercise 1.2.82. On the other hand, of course, we can assign names to anonymous functions:

```
> let dec = \n -> n - 1
```

This means exactly the same as if we had written

```
> let dec n = n - 1
```

In fact, the latter is automatically translated into the former by the compiler. The latter syntax is provided as so-called *syntactic sugar*, i.e., a mechanism to make programming a little more convenient.

1.2.16 Higher-order functions

Exercise 1.2.83. By now, you may (rightfully) ask what the big use of anonymous functions is at all, when all we might really want to do in the end is to assign a name to them again.

The answer lies – again – in the fact that functions in Haskell are more versatile than in many other languages. In particular, functions can be arguments to other functions. A highly useful example of such a function is `map`:

```
> map dec [1, 2, 3]
[0, 1, 2]
```

As you can see, `dec` (from Exercise 1.2.82) is applied to every element of the list. That's what `map` does: it takes a function, and a list, and it applies the function to every element in that list. It is very similar to an iterator in many other languages.

A function such as `map` that takes other functions as arguments, is called a *higher-order function*.

Exercise 1.2.84. For a function such as `map`, it turns out to be incredibly useful that we don't have to assign a name to every argument function. Try the following examples:

```
> map (\s -> " " ++ s) ["Hello", "world"]
> map (\n -> n `mod` 2) [1, 2, 3, 4, 5, 6, 7]
> map (\n -> n * n) [1, 2, 3, 4, 5, 6, 7]
> map (take 2) ["Hello", "world"]
> map not [True, False, True]
> map (\s -> (reverse (take 2 (reverse s)))) ["Hello", "world"]
```

Think of more examples and try them out. Can you imagine what the type of `map` is? If you have a guess, try

```
> :t map
```

Did you guess right? If yes, you can be really proud of yourself. If not, no problem, just look at the examples again and try to understand the type.

Exercise 1.2.85. Another higher-order function is `filter`. It takes a test, that is a function producing a Boolean value and applies it to all elements of a list. It only keeps the list elements for which the test evaluates to `True`. Here are a few examples (recall Exercise 1.2.68):

```
> filter even [1, 2, 3, 4, 5, 6]
> filter (\x -> not (null x)) ["a", "list", "", "of", "strings", ""]
> filter (\x -> x >= 3) [1, 7, 3, 5, 4, 2]
> filter not [True, False, True]
> filter (\x -> x + 1) [1, 2, 3, 4, 5, 6]
```

The operator (>=) just compares if a number is at least as large as the other. The last example will produce a type error. This situation may arise in a real program if you confuse `map` and `filter`, which are similar problems. Can you explain the type error? Look at the type of `filter` and at the type of the anonymous function for help:

```
> :t filter
> :t \x -> x + 1
```

Exercise 1.2.86. What can you say about this function:

```
> let myfilter = filter (\x -> x True)
```

What is its type? How does it behave? Is it useful?

Exercise 1.2.87. Look at types of the following higher-order functions. Apply arguments of the appropriate types to discover what these functions are doing:

```
dropWhile
takeWhile
all
any
```

Exercise 1.2.88. Another higher-order function – in fact, an operator – is *function composition*, written as a dot in Haskell:

```
> :t (.)
(:) :: (b -> c) -> (a -> b) -> a -> c
```

That's a complicated type, so let's rather look at the definition. In fact, we can define it at the interpreter prompt ourselves (thereby shadowing the already existing definition):

```
> let (.) f g = \x -> f (g x)
```

This operator takes two functions `f` and `g`, and it results in the function that, given an `x`, first applies `g` and then `f` to it. Try the following examples:

```
> (tail . tail) "Hello"
> map (head . tail . tail) ["Hello", "world"]
> ((\x -> x + 1) . (\x -> x * 3)) 42
```

Function composition can thus be used to sequence functions. The rightmost function in the sequence is applied first.

1.2.17 Operator sections

Exercise 1.2.89. The final expression in Exercise 1.2.88 can actually be written much simpler. Try

```
> ((+1) . (*3)) 42
```

Similarly, try

```
> map (+1) [1..5]
```

An operator can be partially applied to its first or second argument if placed in parentheses. This construct is called an *operator section* and is a convenient form of syntactic sugar. In particular, `(+1)` is the same as writing `\x -> x + 1`.

Try also

```
> map (2^) [1..5]
> map (^2) [1..5]
> map (10-) [1..5]
> map (-10) [1..5]
```

and analyze the result. The final expression will cause a type error, because unfortunately, `(-10)` is not interpreted as an operator section, but as the negative number `10`. In this case, you have to write

```
> map (\x -> x - 10) [1..5]
```

Exercise 1.2.90 (medium)**.** Try to predict the result of

```
> map ((-) 10) [1..5]
```

and verify your prediction.

Exercise 1.2.91. Operator sections also work for backquoted function names. Try:

```
> map (‘mod‘3) [1..10]
```

Exercise 1.2.92. Of course, operator sections are just normal expressions, so they have the same type as the anonymous functions they represent. Verify, for example, that

```
> :t (+1)
```

and

```
> :t \x -> x + 1
```

lead to the same result.

1.2.18 Loading modules

Exercise 1.2.93. So far, we have evaluated Haskell *expressions* in the interactive interpreter. We have also used `let` *statements* to assign names to functions. Furthermore, we have used special interpreter instructions (starting with a colon) to get more information.

We are now at a point where we move to full Haskell *programs*. Programs can contain expressions, but also so-called *top-level declarations*, for instance to declare modules, the visibility of functions, type classes or datatypes, and of course functions. Such top-level declarations cannot be entered interactively. On the other hand, interpreter commands such as `:t` are not part of the Haskell language and cannot be used in a program.

The simplest useful program consists of a single function. Open any text editor and create a text file called `Lab.hs`. Enter the following line:

```
inc x = x + 1
```

Then save and close the file. The file now contains the definition of a single function named `inc`, that adds `1` to its argument. Note that in a program, no `let` is required (and not even allowed) before the name of the function, but apart from that, the declaration of `inc` is equivalent to typing

```
> let inc x = x + 1
```

at the GHCi prompt.

As you will see, you have more possibilities to write programs within a file than interactively at the GHCi prompt. But the good thing is that you can still *test* your programs in GHCi. To make GHCi aware of your file, you have two possibilities: first, you can type

```
> :l Lab.hs
```

if the file `Lab.hs` is in your current working directory (otherwise you can give a full path name to that file). The command `:l` (for *load*) then loads the file into the interpreter.

If you start a new GHCi session, you can also pass the name of the file you're interested in as an argument to the GHCi command on the command line. Using the command line argument is more convenient if you are typically working with a command line. If you are starting GHCi by clicking on an icon, using the `:l` command is more convenient.

After loading the source file, verify that you can use the function `inc` defined therein by typing

```
> inc 41
```

2 Smaller per topic exercises

2.1 Tooling (*)

Exercise 2.1.1 (Cabal). Submit your solutions to the assignments as a Cabal package. Turn the code (for the `perms_smooth` task, see exercise 2.7.1) into a library. Include the solutions to the theoretical packages as extra documentation files. Write a suitable package description, and include a Setup script so that the package can easily be built and installed using Cabal. As a package name, choose a name that includes your CS logins. Produce the file using the command `cabal sdist`.

2.2 Programming

Exercise 2.2.1 ((Tail) recursion). [15%] Consider the datatype

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

The function `splitleft` splits off the leftmost entry of the tree and returns that entry as well as the remaining tree:

```
splitleft :: Tree a -> (a, Maybe (Tree a))
splitleft (Leaf a)   = (a, Nothing)
splitleft (Node l r) = case splitleft l of
                         (a, Nothing) -> (a, Just r)
                         (a, Just l') -> (a, Just (Node l' r))
```

Write a *tail-recursive* variant of `splitleft`.

Hint. Generalize `splitleft` by introducing an additional auxiliary parameter. Recall that functions can be used as parameters. If you do not know what "tail-recursive" means, look up the definition somewhere.

Exercise 2.2.2 (Tree unfold). Recall `unfoldr`:

```
unfoldr :: (s -> Maybe (a, s)) -> s -> [a]
unfoldr next x =
  case next x of
```

```
    Nothing    -> []
    Just (y,r) -> y:unfoldr next r
```

We can define an unfold function for trees as well:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
  deriving Show

unfoldTree :: (s -> Either a (s,s)) -> s -> Tree a
unfoldTree next x =
  case next x of
    Left y      -> Leaf y
    Right (l,r) -> Node (unfoldTree next l) (unfoldTree next r)
```

Task. Define the following functions using `unfoldr` or `unfoldTree`:

```
iterate :: (a -> a) -> a -> [a]
```

(The call `iterate f x` generates the infinite list `[x,f x,f (f x),...]`.)

```
map :: (a -> b) -> [a] -> [b]
```

(As defined in the prelude.)

```
balanced :: Int -> Tree ()
```

(Generates a balanced tree of the given height.)

```
sized :: Int -> Tree Int
```

(Generates any tree with the given number of nodes. Each leaf should have a unique label.)

Exercise 2.2.3 (Use of fix)**.** Given the function

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

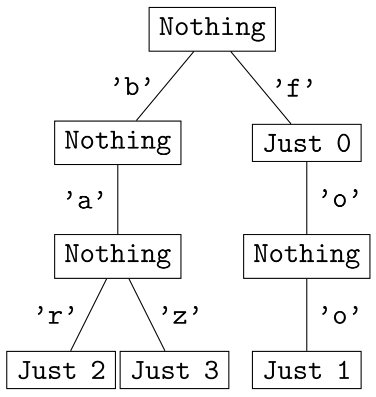
Define the function `foldr` as an application of `fix` to a term that is not recursive.

Exercise 2.2.4 (Trie)**.** A *trie* (sometimes called a prefix tree) is an implementation of a finite map for structured key types where common prefixes of the keys are grouped together. Most frequently, tries are used with strings as key type. In Haskell, such a trie can be represented as follows:

```
data Trie a = Node (Maybe a) (Map Char (Trie a))
```

A node in the trie contains an optional value of type `a`. If the empty string is in the domain of the trie, the associated value can be stored here. Furthermore, the trie maps single characters to subtries. If a key starts with one of the chracters contained in the map, then the rest of the key is looked up in the corresponding subtrie.

The following picture shows an example trie that maps `"f"` to `0`, `"foo"` to `1`, `"bar"` to `2` and `"baz"` to `3`:



The implementation should obey the following invariant: if a trie (or subtrie) is empty, i.e., if it contains no values, it should always be represented by

```
Node Nothing Data.Map.empty
```

Task. Write a module `Data.Trie` containing a datatype of tries as above and the following functions:

```
empty  :: Trie a
null   :: Trie a -> Bool
valid  :: Trie a -> Bool
insert :: String -> a -> Trie a -> Trie a
lookup :: String -> Trie a -> Maybe a
delete :: String -> Trie a -> Trie a
```

The function `valid` tests if the invariant holds. All operations should maintain the invariant.

Exercise 2.2.5 (Type hiding)**.** Define a function `count` such that the following program is well-typed

```
test :: [Int]
test = [count, count 1 2 3, count "" [True, False] id (+)]
```

and evaluates to `[0, 0, 0]`. In other words, `count` should accept an arbitrary number of arguments (of arbitrary types), and just always return `0`.

Then redefine the function `count` such that `test` evaluates to `[0, 3, 4]`, i.e., `count` should return the number of arguments.

Please submit both versions of the function.

Hint: no language extensions are required to solve this exercise.

Exercise 2.2.6 (Partial computation)**.** The type constructor `Partial` can be used to describe possibly nonterminating computations in such a way that they remain productive.

```
data Partial a = Now a | Later (Partial a)
  deriving Show
```

We can now describe a productive infinite loop as follows:

```
loop = Later loop
```

Even functions that terminate can produce a `Later` for every step they perform, allowing us to judge the complexity of the computation afterwards by looking at how many `Later` occurrences there are before the result.

```
runPartial :: Int -> Partial a -> Maybe a
runPartial _ (Now x)   = Just x
runPartial 0 (Later p) = Nothing
runPartial n (Later p) = runPartial (n - 1) p
```

Using `runPartial`, we can then run a partial computation and give a bound on the number of steps it is allowed to perform. If no result is delivered in the allowed number of steps, `Nothing` is returned. Note that `runPartial n loop` terminates for all non-negative choices of `n`.

There also is

```
unsafeRunPartial :: Partial a -> a
unsafeRunPartial (Now x)   = x
unsafeRunPartial (Later p) = unsafeRunPartial p
```

that runs a partial computation with the risk of nontermination.

The `Partial` type constructor forms a monad:

```
instance Monad Partial where
  return      = Now
  Now x   >>= f = f x
  Later p >>= f = Later (p >>= f)
```

The function `tick` introduces an explicit delay:

```
tick = Later (Now ())
```

Here is another example:

```
psum :: [Int] -> Partial Int
psum xs = liftM sum (mapM (\x -> tick >> return x) xs)
```

Try psum on a couple of lists to see what it does.

It also forms a MonadPlus:

```
class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Here, mplus gives us a form of addition (choice) between two partial computations. The idea is that both computations are run in parallel, and the one terminating earlier is returned. The neutral element of this form of choice is loop, the function that never returns.

```
instance MonadPlus Partial where
  mzero = loop
  mplus = merge

merge :: Partial a -> Partial a -> Partial a
merge (Now x)   _         = Now x
merge _         (Now x)   = Now x
merge (Later p) (Later q) = Later (merge p q)
```

Use this monad as well as mzero and mplus (or derived functions) to define a function

```
firstsum :: [[Int]] -> Partial Int
```

that performs psum on every of the integer lists and returns the result that can be obtained with as few delays as possible.

Example:

```
runPartial 100 $ firstsum [repeat 1, [1, 2, 3], [4, 5], [6, 7, 8], cycle [5, 6]]
```

returns Just 9.

Unfortunately, firstsum will not work on infinite (outer) lists and

```
runPartial 200 $ firstsum (cycle [repeat 1, [1, 2, 3], [4, 5], [6, 7, 8], cycle [5, 6]])
```

will loop. The problem is that merge schedules each of the alternatives in a fair way. When using merge on an infinite list, all computations are evaluated one step before the first Later is produced. The solution is to write an unfair merge. Rewrite merge such that the above test returns Just 9 and also

```
runPartial 200 $ firstsum (replicate 100 (repeat 1) ++ [[1]] ++ repeat (repeat 1))
```

returns Just 1.

Exercise 2.2.7 (Object-orientation). Using open recursion and an explicit fixed-point operator similar to

```
fix f = f (fix f)
```

we can simulate some features commonly found in OO languages in Haskell. In many OO languages, objects can refer their own methods using the identifier `this`, and to methods from a base object using `super`.

We model this by abstracting from both `this` and `super`:

```
type Object a = a -> a -> a
data X = X {n :: Int, f :: Int -> Int}
x, y, z :: Object X
x super this = X {n = 0, f = \i -> i + n this}
y super this = super {n = 1}
z super this = super {f = f super . f super}
```

We can extend one "object" by another using `extendedBy`:

```
extendedBy :: Object a -> Object a -> Object a
extendedBy o1 o2 super this = o2 (o1 super this) this
```

By extending an object o_1 with an object o_2, the object o_1 becomes the super object for o_2.

Once we have built an object from suitable components, we can close it to make it suitable for use using a variant of `fix`:

```
fixObject o = o (error "super") (fixObject o)
```

We close the object `o` by instantiating it with an error super object and with itself as `this`.

Look at what the type of `fixObject` is and familiarize yourself with the behaviour of `fixObject` by trying the following expressions:

```
n (fixObject x)
f (fixObject x) 5
n (fixObject y)
f (fixObject y) 5
n (fixObject (x `extendedBy` y))
f (fixObject (x `extendedBy` y)) 5
f (fixObject (x `extendedBy` y `extendedBy` z)) 5
f (fixObject (x `extendedBy` y `extendedBy` z `extendedBy` z)) 5
```

Task. Define an object

```
zero :: Object a
```

such that for all types `t` and objects `x :: Object t`, the equation `x 'extendedBy' zero` ≡ `zero 'extendedBy' x` ≡ `x` hold. (You do not need to provide the proof.)

A more interesting use for these functional objects is for adding effects to functional programs in an aspect-oriented way.

In order to keep a function extensible, we write it as an object, and keep the result value monadic:

```
fac :: Monad m => Object (Int -> m Int)
fac super this n =
  case n of
    0 -> return 1
    n -> liftM (n*) (this (n - 1))
```

Note that recursive calls have been replaced by calls to `this`. We can now write a separate aspect that counts the number of recursive calls:

```
calls :: MonadState Int m => Object (a -> m b)
calls super this n =
  do
    modify (+1)
    super n
```

We can now run the factorial function in different ways:

```
runIdentity (fixObject fac 5)                        ≡ 120
runState    (fixObject (fac 'extendedBy' calls) 5) 0 ≡ (120, 6)
```

Task. Write an aspect `trace` that makes use of a writer monad to record whenever a recursive call is entered and whenever it returns. Also give a type signature with the most general type. Use a list of type

```
data Step a b = Enter a
              | Return b
  deriving Show
```

to record the log. As an example, the call

```
runWriter (fixObject (fac 'extendedBy' trace) 3)
```

yields

```
(6, [Enter 3, Enter 2, Enter 1, Enter 0, Return 1, Return 1, Return 2, Return 6])
```

Exercise 2.2.8 (Enumeration)**.** Consider the following datatype:

```
data GP a = End a
          | Get (Int -> GP a)
          | Put Int (GP a)
```

A value of type `GP` can be used to describe programs that read and write integer values and return a final result of type `a`. Such a program can end immediately (`End`). If it reads an integer, the rest of the program is described as a function depending on this integer (`Get`). If the program writes an integer (`Put`), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

```
echo = Get (\n -> Put n echo)
```

Task. Write a function

```
run :: GP a -> IO a
```

that can run a `GP`-program in the `IO` monad. A `Get` should read an integer from the console, and `Put` should write an integer to the console.

Here is an example run from GHCi:

```
> run echo
? 42
42
? 28
28
? 1
1
? -5
-5
? Interrupted.
>
```

[To better distinguish inputs from outputs, this version of `run` prints a question mark when expecting an input.]

Task. Write a `GP`-program `add` that reads two integers, writes the sum of the two integers, and ultimately returns `()`.

Task. Write a `GP`-program `accum` that reads an integer. If the integer is `0`, it returns the current total. If the integer is not `0`, it adds the integer to the current total, prints the current total, and starts from the beginning.

Task. Instead of running a GP-program in the IO monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

```
simulate :: GP a -> [Int] -> (a, [Int])
```

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated.

A map function for GP can be defined as follows:

```
instance Functor GP where
  fmap f (End x)   = End (f x)
  fmap f (Get g)   = Get (fmap f . g)
  fmap f (Put n x) = Put n (fmap f x)
```

Task. Define sensible instances of Monad and MonadState for GP. How is the behaviour of the MonadState instance for GP different from the usual State type?

Exercise 2.2.9 (Idiom, continuation passing)**.** Find Haskell definitions for the functions start, stop, store, add and mul such that you can embed a stack-based language into Haskell:

```
p1, p2, p3 :: Int
p1 = start store 3 store 5 add stop
p2 = start store 3 store 6 store 2 mul add stop
p3 = start store 2 add stop
```

Here, p_1 should evaluate to 8 and p_2 should evaluate to 15. The program p_3 is allowed to fail at runtime.

Once you have that, try to find a solution that rejects programs that require non-existing stack elements during type checking.

Hint: Type classes are *not* required to solve this assignment. This is somewhat related to continuations. Try to first think about the types that the operations should have, then about the implementation.

2.3 Monads

The goal of this task is to create your own *monad*. In fact, you have to extend the well-known state monad (see Control.Monad.State in the hierarchical libraries), and add some extra features to it.

The first step is to introduce a type constructor for the new monad:

```
data StateMonadPlus s a = ...
```

The type variables `s` and `a` have the standard meaning: `s` is the type of the state to carry and `a` is the type of the return value. Make the new monad an instance of the `MonadState` type class. Hence, you have to include:

```
import Control.Monad.State
```

We now discuss the three additional features that your monad has to support.

Feature 1: Diagnostics

We want to gather information about the number of calls to all primitive functions that work on a `StateMonadPlus`. For this, you have to write a function `diagnostics` with the following type:

```
diagnostics :: StateMonadPlus s String
```

This function should count the number of binds (>>=) and `returns` (and other *primitive* functions) that have been encountered, including the call to `diagnostics` at hand. Secondly, provide a function

```
annotate :: String -> StateMonadPlus s a -> StateMonadPlus s a
```

which allows a user to annotate a computation with a given label. The functions for Features 2 and 3, as well as `get` and `put`, should also be part of the diagnosis.

As an example, consider the input

```
do return 3 >> return 4
   return 5
   diagnostics
```

which should return the string

```
"[bind=3, diagnostics=1, return=3]"
```

Note that >> is implemented in terms of >>=, and thus also counts as a bind.

Here is another example:

```
do annotate "A" (return 3 >> return 4)
   return 5
   diagnostics
```

This returns the string

```
"[A=1, bind=3, diagnostics=1, return=3]"
```

Feature 2: Failure

A second feature of your monad is that it can fail during a computation. The `Monad` type class offers the following member function:

```
fail :: (Monad m) => String -> m a
```

Calling this function should not result in an exception. To facilitate this, the function `runStateMonadPlus` (which will be explained later) returns an `Either` value: `Left` indicates that the computation failed, `Right` indicates success. You may have to change the `StateMonadPlus` data type to cope with this.

Feature 3: History of states

The last feature is to save the current state, and to restore a previous state as the current state. Include the following type class definition in your code, and make `StateMonadPlus` an instance of this type class.

```
class MonadState s m => StoreState s m | m -> s where
  saveState :: m ()
  loadState :: m ()
```

The part `m -> s` in the class declaration is a *functional dependency*, indicating that the type `s` is uniquely determined by the choice of `m`. Functional dependencies limit the instance declarations that are valid, and in turn allow the type checker to make use of the functional dependency while inferring type. Functional dependencies are a Haskell language extension. To enable it, you must put a language pragma at the top of your module:

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

Saving and loading states should be implemented as a stack: saving a state means pushing the current state on the stack, while loading a state means popping a state from the stack to replace the current one. If `loadState` is called with an empty stack, then the computation in the monad should fail (as explained for Feature 2).

Here is an example expression:

```
do i1 <- get; saveState
   modify (*2)
   i2 <- get; saveState
   modify (*2)
   i3 <- get; loadState
   i4 <- get; loadState
   i5 <- get
   return (i1, i2, i3, i4, i5)
```

This program should return the value `(1, 2, 4, 2, 1)` if we start with the state consisting of the integer `1`.

Running the monad

You have to write a function

```
runStateMonadPlus :: StateMonadPlus s a -> s -> Either String (a, s)
```

for running the monad. Given a computation in the `StateMonadPlus` and an initial state, `runStateMonadPlus` returns either an error message if the computation failed, or the result of the computation and the final state.

To turn your module into a proper library, you should also think about which functions should be exposed to outside this module, and which functions should be hidden (and only be visible inside the current module). You might want to consider re-exporting all functionality offered by the `Control.Monad.State` module.

Try also to define a number of unit tests or even QuickCheck properties.

Bonus questions

Exercise 2.3.1. Do the monad laws hold for `StateMonadPlus`? Explain your answer.

Exercise 2.3.2. What are the advantages of hiding (constructor) functions? How important is this for each of the three additional features supported by `StateMonadPlus`?

Exercise 2.3.3. What are the modifications required to make a monad transformer for `StateMonadPlus`?

Exercise 2.3.4. Suppose that we want to write a function

```
diagnosticsFuture :: StateMonadPlus s String
```

which provides information about the computations in `StateMonadPlus` that are still to come. Explain how this would affect your code. If you feel that such a facility cannot be implemented, then you should give some arguments for your opinion. If you believe it can be done, then try to do so.

2.4 Programming jointly with types and values

Exercise 2.4.1 (Fixpoint)**.** The lambda term

```
y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

(that encodes a fixed point combinator in the untyped lambda calculus) does not type check in Haskell. Try it! Interestingly though, recursion on the type level can be used to introduce recursion on the value level. If we define the recursive type

```
data F a = F {unF :: F a -> a}
```

then we can "annotate" the definition of `y` with applications of `F` and `unF` such that `y` typechecks. Do it!

Exercise 2.4.2 (Nested datatype)**.** Here is a nested datatype for square matrices:

```
type Square a    = Square' Nil a
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a    = Nil
data Cons t a = Cons a (t a)
```

Give Haskell code that represents the following two square matrices as elements of the `Square` datatype:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Note: you don't have to define any functions for the `Square` datatype. Defining sensible functions for `Square` (even `show`) is not entirely trivial and is the topic of other assignments.

Exercise 2.4.3 (Nested datatype)**.** Recall the datatype of square matrices:

```
type Square      = Square' Nil
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a    =  Nil
data Cons t a =  Cons a (t a)
```

Note that I have eta-reduced the definition of `Square`. This turns out to be necessary in the end where I'll mention it again.

Let's investigate how we can derive an equality function on square matrices. We do so very systematically by deriving an equality function for each of the four types. We follow a simple, yet powerful principle: type abstraction corresponds to term abstraction, and type application corresponds to term application.

What does this mean? If a type `f` is parameterized over an argument `a`, then in general, we have to know how equality is defined on `a` in order to define equality on `f a`. Therefore we define

```
eqNil :: (a -> a -> Bool) -> (Nil a -> Nil a -> Bool)
eqNil eqA Nil Nil = True
```

In this case, the `a` is not used in the definition of `Nil`, so it is not surprising that we do not use `eqA` in the definition of `eqNil`. But what about `Cons`? The datatype `Cons` has two arguments `t` and `a`, so we expect two arguments to be passed to `eqCons`, something like

```
eqCons eqT eqA (Cons x xs) (Cons y ys) = eqA x y && ...
```

But what should the type of `eqT` be? The `t` is of kind `* -> *`, so it can't be `t -> t -> Bool`. We can argue that we should use `t a -> t a -> Bool`, because we use `t` applied to `a` in the definition of `Cons`. However, a better solution is to recognise that, being a type constructor of kind `* -> *`, an equality function on `t` should take an equality function on its argument as a parameter. And, moreover, it does not matter what this parameter is! A function like `eqNil` is polymorphic in type `a`, so let us require that `eqT` is polymorphic in the argument type as well:

```
eqCons :: (∀b . (b -> b -> Bool) -> (t b -> t b -> Bool)) ->
          (a -> a -> Bool) ->
          (Cons t a -> Cons t a -> Bool)
eqCons eqT eqA (Cons x xs) (Cons y ys) = eqA x y && eqT eqA xs ys
```

Now you can see how we apply `eqT` to `eqA` when we want equality at type `t a` – the type application corresponds to term application.

Task. A type with a ∀ on the inside requires the extension `RankNTypes` to be enabled. Try to understand what the difference is between a function of the type of `eqCons` and a function with the same type but the ∀ omitted. Can you omit the ∀ in the case of `eqCons` and does the function still work?

Now, on to `Square'`. The type of `eqSquare'` follows exactly the same idea as the type of `eqCons`:

```
eqSquare' :: (∀b . (b -> b -> Bool) -> (t b -> t b -> Bool)) ->
             (a -> a -> Bool) ->
             (Square' t a -> Square' t a -> Bool)
```

We now for the first time have more than one constructor, so we actually have to give multiple cases. Let us first consider comparing two applications of `Zero`:

```
eqSquare' eqT eqA (Zero xs) (Zero ys) = eqT (eqT eqA) xs ys
```

Note how again the structure of the definition follows the structure of the type. We have a value of type `t (t a)`. We compare it using `eqT`, passing it an equality function for values of type `t a`. How? By using `eqT eqA`.

The remaining cases are as follows:

```
eqSquare' eqT eqA (Succ xs) (Succ ys) = eqSquare' (eqCons eqT) eqA xs ys
eqSquare' eqT eqA _ _                 = False
```

The idea is the same – let the structure of the recursive calls follow the structure of the type.

Task. Again, try removing the ∀ from the type of `eqSquare'`. Does the function still typecheck? Try to explain!

Now we're done:

```
eqSquare :: (a -> a -> Bool) -> Square a -> Square a -> Bool
eqSquare = eqSquare' eqNil
```

Test the function. We can now also give an `Eq` instance for `Square` – this requires the minor language extension `TypeSynonymInstances`, because for some stupid reason, Haskell 98 does not allow type synonyms like `Square` to be used in instance declarations:

```
instance Eq a => Eq (Square a) where
  (==) = eqSquare (==)
```

Task. Systematically follow the scheme just presented in order to define a `Functor` instance for square matrices. I.e., derive a function `mapSquare` such that you can define

```
instance Functor Square where
  fmap = mapSquare
```

This instance requires `Square` to be defined in eta-reduced form in the beginning, because Haskell does not allow partially applied type synonyms.

Exercise 2.4.4 (Nested datatype)**.** Haskell does not allow partially applied type synonyms.

Recall that in previous assignments, we defined

```
type Square = Square' Nil
```

and not

```
type Square a = Square' Nil a
```

With the former definition (and enabled `TypeSynonymInstances`), we can make `Square` an instance of class `Functor`, with the latter definition, we cannot.

Why is this restriction in place? Try to find problems arising from partially applied type synonyms, and describe them (as concisely as possible) with a few examples.

2.5 Programming with classes

Exercise 2.5.1 (Split)**.** Consider the following class

```
class Splittable a where
  split :: a -> (a, a)
```

for types that allow values to be split. Random number generators (for instance `StdGen`) allow such a split operation:

```
instance Splittable StdGen where
  split = System.Random.split
```

We can also make other types an instance of `Splittable`. Define an instance `Splittable [a]` where, assuming that the list passed is infinite, the list is split into one list containing all the odd-indexed elements, and one containing all the even-indexed elements of the original list.

Define an instance `Splittable Int` where `n` is split into `2 * n` and `2 * n + 1`.

Consider the datatype

```
data SplitReader r a = SplitReader {runSplitReader :: r -> a}
```

which is isomorphic to the `Reader` datatype. Define a variant of the `Reader` monad

```
instance (Splittable r) => Monad (SplitReader r)
```

where the passed state is split before it is passed on. Also implement the instance of `MonadReader`:

```
instance (Splittable r) => MonadReader r (SplitReader r)
```

You have to pass enable the `FlexibleInstances` and `MultiParamTypeClasses` language extensions to make GHC accept this instance. The methods of the class `MonadReader` are

```
ask :: (MonadReader r m) => m r
```

that allows you to access the read state, and

```
local :: (MonadReader r m) => (r -> r) -> m a -> m a
```

that allows you to locally modify the read state.

Finally, consider the function

```
labelTree :: Int -> SplitReader Int (Tree Int)
labelTree 0 = return Leaf
labelTree n = return () >> liftM3 Node (labelTree (n - 1)) ask (labelTree (n - 1))
```

where

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
  deriving Show
```

When calling `runSplitReader (labelTree 3) 1`, the function returns

```
Node (Node (Node Leaf 214 Leaf) 54  (Node Leaf 886  Leaf))
     14
     (Node (Node Leaf 982 Leaf) 246 (Node Leaf 3958 Leaf))
```

Is this what you expected? If you remove `return () ≫` in the definition of `labelTree` and try again, what happens? What do these results imply?

Exercise 2.5.2 (Context reduction)**.** Consider the following system of classes.

```
class A a
class (A a) ⇒ B a
instance A Bool
instance B Bool
instance A a ⇒ A (Maybe a)
instance (A a, B a) ⇒ A [a]
```

Prove `B Int` ⊩ `A (Maybe (Maybe Int))` and ∅ ⊩ `A (Maybe [Bool])` using the general entailment rules as well as (super) and (inst) from Slides 10-11 to 10-15. Draw proof trees such as on Slide 10-17.

Exercise 2.5.3 (Evidence translation)**.** Consider the following module:

```
import Control.Monad.Reader
import System.Random

one :: Int
one = 1

two :: Int
two = 2

randomN :: (RandomGen g) ⇒ Int -> g -> Int
randomN n g = (fst (next g) ‘mod‘ (two * n + one)) - n

sizedInt = do
             n <- ask
             g <- lift ask
             return (randomN n g)
```

What is the most general type of `sizedInt`? (Note that type inference may not work for `sizedInt`, but you should be able to explain the error message by now and know how to fix it. Note further that the most general type will only be accepted by GHC in a type signature when `FlexibleContexts` are enabled.)

Assuming this most general type, perfom an evidence translation for all the overloading involved in the functions `randomN` and `sizedInt`. First, define the record types for the classes involved. You can *ignore* the fact that literals and arithmetic operations are overloaded and just use `one` and `two` as monomorphic integers. You only have to include those methods in the records that are actually used in the program above. Hoever, you *should* consider the desugaring (you may simplify and ignore the `let` statements for the patterns) of the `do` notation to the monad operations, as well as the overloaded `ask` and `lift` functions. In order to define the record types correctly, you must enable the `PolymorphicComponents` or `RankNTypes` language extensions to allow polymorphic fields in datatypes.

Then translate `randomN` and `sizedInt` similar to the translation on Slide 10-21. You are allowed to introduce local abbreviations using `let` and `where` for often-used expressions. The resulting program must, of course, still be type correct in Haskell.

2.6 Type extensions

Exercise 2.6.1 (GADT). Here is a datatype of contracts:

```
data Contract :: * -> * where
  Pred :: (a -> Bool) -> Contract a
  Fun  :: Contract a -> Contract b -> Contract (a -> b)
```

The datatype is a Generalized Algebraic Datatype (GADT). The constructors do not both target any contract `Contract a`, but the `Fun` constructor has a restricted result type, i.e., it can only construct function contracts. GADTs require the language flag/pragma `GADTs`.

A contract can be a predicate for a value of arbitrary type. For functions, we offer contracts that contain a precondition on the arguments, and a postcondition on the results.

Contracts can be attached to values by means of `assert`. The idea is that `assert` will cause run-time failure if a contract is violated, and otherwise return the original result:

```
assert :: Contract a -> a -> a
assert (Pred p)       x = if p x then x else error "contract violation"
assert (Fun pre post) f = assert post . f . assert pre
```

For function contracts, we first check the precondition on the value, then apply the original function, and finally check the postcondition on the result. Note that the case for `Fun` makes use of the fact that the `Fun` constructor targets only function contracts. Because of this knowledge, GHC allows us to apply `f` as a function.

For example, the following contract states that a number is positive:

```
pos :: (Num a, Ord a) ⇒ Contract a
pos = Pred (>0)
```

We have

```
assert pos 2 ≡ 2
assert pos 0 ≡ ⊥       (contract violation error)
```

Task. Define a contract

```
true :: Contract a
```

such that for all values `x`, the equation `assert true x ≡ x` holds. Prove this equation using equational reasoning.

Often, we want the postcondition of a function to be able to refer to the actual argument that has been passed to the function. Therefore, let us change the type of `Fun`:

```
DFun :: Contract a -> (a -> Contract b) -> Contract (a -> b)
```

The postcondition now depends on the function argument.

Task. Adapt the function `assert` to the new type of `DFun`.

Task. Define a combinator

```
(—▹) :: Contract a -> Contract b -> Contract (a -> b)
```

that reexpresses the behaviour of the old `Fun` constructor in terms of the new and more general one.

Task. Define a contract suitable for the list index function `(!!)`, i.e., a contract of type

```
Contract ([a] -> Int -> a)
```

that checks if the integer is a valid index for the given list.

Task. Define a contract

```
preserves :: Eq b ⇒ (a -> b) -> Contract (a -> a)
```

where `assert (preserves p) f x` fails if and only if the value of `p x` is different from the value of `p (f x)`. Examples:

```
assert (preserves length) reverse "Hello"        ≡ "olleH"
assert (preserves length) (take 5) "Hello"       ≡ "Hello"
assert (preserves length) (take 5) "Hello world" ≡ ⊥
```

Task. Consider

```
preservesPos  = preserves (>0)
preservesPos' = pos ↠ pos
```

Is there a difference between `assert preservesPos` and `assert preservesPos'`? If yes, give an example where they show different behaviour. If not, try to prove their equality using equational reasoning.

We can add another contract constructor:

```
List :: Contract a -> Contract [a]
```

The corresponding case of `assert` is as follows:

```
assert (List c) xs = map (assert c) xs
```

Task. Consider

```
allPos  = List pos
allPos' = Pred (all (>0))
```

Describe the differences between `assert allPos` and `assert allPos'`, and more generally between using `List` versus using `Pred` to describe a predicate on lists. (Hint: Think carefully and consider different situations before giving your answer. What about using the `allPos` and `allPos'` contracts as parts of other contracts? What about lists of functions? What about infinite lists? What about strict and non-strict functions working on lists?)

2.7 Performance

Exercise 2.7.1 (Algorithm design)**.** Given the following type signature

```
smooth_perms :: Int -> [Int] -> [[Int]]
```

for a function which returns all permutations of its second argument for which the distance between each two successive elements is at most the first argument.

```
split [] = []
split (x:xs) = (x, xs):[(y, x:ys) | (y, ys) <- split xs]
perms [] = [[]]
perms xs = [(v:p) | (v, vs) <- split id xs, p <- perms vs]
smooth n (x:y:ys) = abs (y - x) ⩽ n && smooth n (y:ys)
smooth _ _ =        True
```

```
smooth_perms :: Int -> [Int] -> [[Int]]
smooth_perms n xs = filter (smooth n) (perms xs)
```

A straightforward solution is to generate all permutations and then to filter out the smooth ones. This however is expensive. A better approach is to build a tree, for which it holds that each path from the root to a leaf correspond to one of the possible permutations, next to prune this tree such that only smooth paths are represented, and finally to use this tree to generate all the smooth permutations from.

Now define this tree data type, a function which maps a list onto this tree, the function which prunes the tree, and finally the function which generates all permutations.

2.8 Observing: performance, testing, benchmarking

Exercise 2.8.1 (Profiling, testing, benchmarking)**.** Elaborate on exercise 2.7.1 by using various observation tools:

- Give a `quickCheck` specification and check, by defining a function `allSmoothPerms`.
- Use the `criterion` package to make and run benchmarks for the given naive solution and your solution, in order to find out whether your solution really gives higher performance.
- Use heap profiles to analyse and draw conclusions about the differences.

Exercise 2.8.2 (Quickcheck)**.** QuickCheck's `Arbitrary` class is defined as follows

```
class Arbitrary a where
  arbitrary :: Gen a
```

The type `Gen` is defined as

```
newtype Gen a = MkGen {unGen :: StdGen -> Int -> a}
```

(These definitions are from QuickCheck-2. The definitions in QuickCheck-1 are slightly different, but essentially the same. It does not matter which version you use.) Look at the QuickCheck source code for the definition of the monad instance. Assemble an equivalent monad from the `Reader` and `SplitReader` monads or monad transformers.

Define the function `sizedInt :: Gen Int` just using `ask`, `lift` and `System.Random.randomR` (i.e., not using the internal structure of the `Gen` type), such that `sizedInt` generates a random number between `-n` and `n` where `n` is the read integer.

Exercise 2.8.3 (Profiling)**.** Generate heap profiles for the following functions:

```
rev  = foldl (flip (:)) []
rev' = foldr (\x r -> r ++ [x]) []
```

by using them as function f in a main program as follows

```
main = print $ f [1..1000000]
```

(adapt the size of 1000000 according to the speed of your machine to get good results). Interpret and try to explain the results!

Do the same for

```
conc xs ys = foldr (:) ys xs
conc'      = foldl (\k x -> k . (x:)) id
```

with

```
main = print $ f [1..1000000] [1..1000000]
```

(where f is conc or conc').

Finally, have a look at

```
f1 = let xs = [1..1000000] in if length xs > 0 then head xs else 0
f2 = if length [1..1000000] > 0 then head [1..1000000] else 0
```

with

```
main = print f
```

(where f is f_1 or f_2).

Remember that the main point of this assignment is not to send in the heap profiles, but to explain them!

Exercise 2.8.4 (Forcing evaluation). Write a function

```
forceBoolList :: [Bool] -> r -> r
```

that completely forces a list of booleans *without using* `seq`. Note that pattern matching drives evaluation.

Explain why the function forceBoolList has the type as specified above and not

```
forceBoolList :: [Bool] -> [Bool]
```

and why seq is defined as it is, and

```
force :: a -> a
force a = seq a a
```

is useless.

Exercise 2.8.5 (Type complexity)**.** A curious fact is that Haskell's type system (even that of Haskell without extensions) has exponential space and time complexity. However, the worst case rarely occurs in practice such that the run-time behaviour of the type checker generally is acceptable. Define a family of Haskell expressions such that the type (i.e., the size of the type expression) grows exponentially in the size of the program. Note that if the type is highly repetitive, the type can internally be represented using sharing. However, different type variables cannot be shared. So, to get a truly large type, you have to try to get as many different type variables as possible. If you find your solution on the internet explain how it works!

2.9 Reasoning (inductive, equational)

Exercise 2.9.1 (Induction over tree)**.** Consider the following definitions:

```
data Tree a = Leaf a                  flatten :: Tree a -> [a]
            | Node (Tree a) (Tree a)flatten (Leaf a)   = [a]
  deriving Show                       flatten (Node l r) = flatten l ++ flatten r

size :: Tree a -> Int                 (++) :: [a] -> [a] -> [a]
size (Leaf a)   = 1                   []     ++ ys = ys
size (Node l r) = size l + size r     (x:xs) ++ ys = x:(xs ++ ys)

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Prove the following Theorem using equational reasoning:

```
∀(t :: Tree a) . length (flatten t) ≡ size t
```

Note that using the induction principle on trees, it is sufficient to show the following two cases:

```
∀(x :: a) . length (flatten (Leaf x)) ≡ size (Leaf x)
```

and

```
∀(l :: Tree a) (r :: Tree a) .
   ( length (flatten l) ≡ size l
  && length (flatten r) ≡ size r
   ) -> length (flatten (Node l r)) ≡ size (Node l r)
```

To prove the Theorem, you will need to prove the following Lemma first:

```
∀(xs :: [a]) (ys :: [a]) . length (xs ++ ys) = length xs + length ys
```

You may use facts about `(+)` such that `(+)` is associative or that `0` is the neutral element of addition.

Exercise 2.9.2 (Isomorphism)**.** Show that the following two types are isomorphic if we ignore the presence of undefined values:

```
type A r a = (r, a -> r -> r)
```

and

```
type B r a = Maybe (a, r) -> r
```

In other words, define functions

```
f :: A r a -> B r a
```

and

```
g :: B r a -> A r a
```

such that `∀(x :: B r a) . f (g x) = x` and `∀(x :: A r a) . g (f x) = x`. Prove these two statements using equational reasoning.

The first proof is an equality between functions. In order to prove that two functions are equal, prove that they return equal results for equal arguments. I.e., instead of the statement given above, prove instead that

```
∀(x :: B r a) (y :: Maybe (a, r)) . f (g x) y = x y
```

For the second proof, you can use *eta-reduction*, another transformation for lambda-terms. If `f` is a function, then `\x -> f x` is equivalent to `f`.

2.10 IO, Files, Unsafety, and the rest of the world (∗∗)

2.10.1 IO Unsafety (1)

Exercise 2.10.1 (IO unsafety)**.** Consider a function of type

```
runIO :: IO a -> a
```

Why is such a function dangerous? (There are several reasons. Try to give example programs that are dangerous or even demonstrate that something strange and unexpected is going on.)

Exercise 2.10.2 (IO)**.** Formulate why the following program is troublesome in Haskell (look at the types).

```
import System.IO.Unsafe
import Data.IORef

main =
  let x = unsafePerformIO (newIORef [])
  in  do
        writeIORef x "abc"
        ys <- readIORef x
        putStrLn (show (ys :: [Int]))
```

Find out how the ML language family (SML, OCaml, F#) prevents this problem

Describe their approach in relation to Haskell's. Keep the explanation short and precise (no more than 60 words).

2.10.2 Server

Exercise 2.10.3 (Chat server)**.** Using STM and the `Network` library, write a simple chat server and client. The server should listen on a particular port (say `9595`) for incoming connections. The client should take the hostname to connect to and a nickname as command line arguments and try to connect to that port. Upon connection, the client should register itself and the nickname with the server. The clients should read messages from the user and display messages from the server. The server should broadcast any message received from any client to all clients (mentioning the nickname). The server should also notify clients about new nicknames joining the chat, and about clients who have left the chat.

Do *not* include any other features in the programs. Instead, try to keep the code as short and concise and elegant as possible.

2.10.3 IO Unsafety (2)

The goal of this task is to show how dangerous it is to have `unsafePerformIO :: IO a -> a` available in the langauge.

The solution to this assignment is easy to find on the internet. So if you want to actually find out yourself, don't look.

Import the following modules

```
import Data.IORef
import System.IO.Unsafe
```

Exercise 2.10.4. Play a bit with the `IORef` functions

```
newIORef
readIORef
writeIORef
```

That implement mutable variables (references) in Haskell.

Exercise 2.10.5. Implement a "running total" program without passing state explicitly, and not using a state monad, but by using a integer reference.

Exercise 2.10.6. Use the function `unsafePerformIO` to define a value

```
anything :: IORef a
```

Can you already see that this looks like a dangerous type?

Exercise 2.10.7. Use `unsafePerformIO` and `anything` to define a function

```
cast :: a -> b
```

that abandons all type safety. Play with `cast` a bit and try things like

```
> cast False :: Int
```

Try to make GHCi crash with a segmentation fault.

2.11 Generic Programming (***)

Exercise 2.11.1 (SYB show & read)**.** Via classes `Show` and `Read` Haskell values can be encoded as strings and read (parsed) back using `show` and `read` respectively. As such `show` and `read` can be used as a poor man's serialization mechanism similar to `encode` and `decode` (from the SYB slides). Look up the implementation of `Show` and `Read` (part of the base package of the Haskell Platform) and re-implement `Show` and `Read` using the `syb` package. Use Quickcheck to test your generic SYB based `show` and `read` implementation against the default `show` and `read`. It may well be that not all corner cases of showing and reading can be covered. If so, explain where and why this happens.

Exercise 2.11.2 (Generic Deriving show & read)**.** Redo exercise 2.11.1 for `read` only, using the generic deriving mechanism available in GHC. Be inspired by the module `Generic.Deriving.Show` for the already implemented `show`.

2.12 Lists (*)

The goal of this assigment is to become more familiar with defining Haskell functions by writing a few functions of your own, mostly on lists.

Sorting

Exercise 2.12.1. Write a function

```
flatten :: [[Int]] -> [Int]
```

that flattens a list of lists of integers to a list of integers.

Example:

```
> flatten [[2, 3], [5, 7, 11]]
[2, 3, 5, 7, 11]
```

Use the standard way to define functions on lists: pattern matching on the constructors. Use recursion in the appropriate place.

Exercise 2.12.2. Find the most general type for `flatten` in the previous assignment, and adapt the type signature accordingle.

Exercise 2.12.3. Write a function

```
insert :: Int -> [Int] -> [Int]
```

that inserts an integer at the right position in an already sorted list.

Example:

```
> insert 5 [2, 3, 7, 11]
[2, 3, 5, 7, 11]
```

Again, use pattern matching on lists to define `insert`. Furthermore, use `if-then-else` or guards to distinguish the important cases.

Exercise 2.12.4. Write a function

```
sort :: [Int] -> [Int]
```

that sorts a list of integers.

Example:

```
> sort [5, 11, 2, 7, 3]
[2, 3, 5, 7, 11]
```

Use the function `insert` from the previous exercise.

Fold

The higher-order function `foldr` is defined as follows:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr cons nil []     = nil
foldr cons nil (x:xs) = cons x (foldr cons nil xs)
```

The function can be used to express a large number of list traversal functions. In particular, `flatten` and `sort` can be written as calls to `foldr`,

As an example, consider the function `add` that adds a list of integers:

```
add :: [Int] -> Int
add []     = 0
add (x:xs) = x + add xs
```

Using `foldr`, we use `0` for `nil` and the `(+)` operator for `cons` and can equivalently write

```
add :: [Int] -> Int
add = foldr (+) 0
```

Exercise 2.12.5. Write `flatten` as an application of `foldr`.

Exercise 2.12.6. Write `sort` as an application of `foldr`.

2.13 Trees (*)

The goal of this task is to work with binary trees. Start a new module and define a datatype of binary trees as follows:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
           deriving Show
```

The `deriving Show` instructs Haskell to automatically generate a function that allows GHCi (and you) to show and print trees on screen.

Exercise 2.13.1. Write a function

```
single :: a -> Tree a
```

that builds a tree with a single element.

Example:

```
> single 3
Node Leaf 3 Leaf
```

Exercise 2.13.2. Write a function

```
size :: Tree a -> Int
```

that counts the number of nodes in a tree.

Example:

```
> size (Node (single 4) 7 (single 5))
3
```

Exercise 2.13.3. Write a function

```
height :: Tree a -> Int
```

that counts the height of a tree (the maximal distance of the root to a leaf).

Example:

```
> height (Node (single 4) 7 (Node (single 1) 3 (single 7)))
4
```

Exercise 2.13.4. Write a function that flattens a tree into a list

```
flatten :: Tree a -> [a]
```

Example:

```
> flatten (Node (single 4) 7 (single 5))
[4,7,5]
```

Exercise 2.13.5. Write a function that reverses a tree

```
reverse :: Tree a -> Tree a
```

Example:

```
> reverse (Node (single 4) 7 (single 5))
Node (Node Leaf 5 Leaf) 7 (Node Leaf 4 Leaf)
```

Exercise 2.13.6 (medium)**.** Write a function that implements tree-sort, an algorithm that is similar to quicksort: given a nonempty list, take the first element, partition the list in all element smaller of equal and all elements larger, and create a node with the first element in the root, and recursively invoke the function build the left and right subtrees. The resulting tree should be a binary search tree.

```
treesort :: Ord a => [a] -> Tree a
```

Calling `flatten` on the result of `treesort` should yield a sorted list.

Exercise 2.13.7. Write a function that checks if a tree is a binary search tree.

```
bst :: Ord a => Tree a -> Bool
```

Exercise 2.13.8 (medium)**.** Write a function that labels a tree, each node differently, by traversing the tree, maintaining a state, and assigning a new number to every node.

```
labelTree' :: Tree a -> State Int (Tree Int)
```

Remember to import `Control.Monad.State` to access the state monad. There is a function

```
runState :: State s a -> s -> (a, s)
```

you can use to pass in an initial state and get at the final state and result in the end:

```
labelTree :: Tree a -> Tree Int
labelTree t = fst (runState (labelTree t))
```

3 Larger programming tasks

3.1 Lists for database representation (*)

In this exercise we will read in a database, perform a simple query on it and present the results to the user in an aesthetically pleasing form. Most exercises can be completed by combing functions from the `Prelude` and the libraries `Data.Char`, `Data.List` and `Data.Maybe` and contain a hint on which functions you could use from these libraries; often a completely different solution, not using these functions, is also possible. A starting framework and the sample database can be found on the Assignments page on the course website.

3.1.1 Parsing

A plain text database consists of a number of lines (each line is called a *row*), with on each line a fixed number of *fieds* separated by a single space. The first row a database table is called the *header* and contains the names of the columns in the table. An example of such a database would be:

```
first last gender salary
Alice Allen female 82000
Bob Baker male 70000
Carol Clarke female 50000
Dan Davies male 45000
Eve Evans female 275000
```

One way of modeling such databases in Haskell would be using the following types:

```
type Field = String
type Row   = [Field]
type Table = [Row]
```

A field is always modeled as a string (even though the database may contain strings that look very much like numbers), a row is a list of fields and a table a list of rows. The head of this list corresponds to the header of the table. (A valid table always has a header and always has at least one column.)

There are several "problems" with this model: for example, it does not enforce that each of the rows in the table must have the same number of fields. However, for the purposes of this first assignment it will suffice. You may assume that all the databases that are presented to program will be well-formed, that is to say, they will always have the same number of fields on each line.

The form in which data is stored inside a file, printed or written on paper, or entered from the keyboard is called its *concrete syntax*. The form in which data is manipulated inside a program is called its *abstract syntax*. The process of transforming some object represented in its concrete syntax into its representation in abstract syntax is called *parsing*.

Exercise 3.1.1. Write a function `parseTable :: [String] -> Table` that parses a table represented in its concrete syntax as a list of strings (each corresponding to a single line in the input) into its abstract syntax. (Hint: use the function `words` from the `Prelude`.)

3.1.2 Pretty printing

In the previous exercise we have seen how we can turn concrete syntax into abstract syntax. The reverse operation—turning abstract syntax into concrete syntax—is often called *pretty printing* or *compilation*. In our case we do not want to convert our abstract syntax into the original concrete syntax, but into a different concrete syntax that is easier to read for humans:

```
+-----+------+-----+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+-----+------+
|Alice|Allen |female| 82000|
|Bob  |Baker |male  | 70000|
|Carol|Clarke|female| 50000|
|Dan  |Davies|male  | 45000|
|Eve  |Evans |female|275000|
+-----+------+-----+------+
```

An apt name for this process might be "prettier printing". Note that we have done several things to make the result look nice:

1. We have made the width of each column exactly as wide as the widest field in this column (including the name in the header).
2. We have added a very fancy looking border around the table, the header and columns.
3. We have typeset the names of the columns in the header in uppercase.
4. We have right-aligned fields that look like (whole) numbers.

Exercise 3.1.2. Write a function `printLine :: [Int] -> String` that, given a list of widths of columns, returns a string containing a horizontal line. For example, `printLine [5, 6, 6, 6]` should return the line `"+-----+------+------+------+"`. (Hint: use the function `replicate`.)

If you can write this function using `foldr` you will get more points for style.

Exercise 3.1.3. Write a function `printField :: Int -> String -> String` that, given a desired width for a field and the contents of a fields, returns a formatted field by adding additional whitespace. If the field only consists of numerical digits, the field should be right-aligned, otherwise it should be left-aligned. (Hint: use the functions `all`, `isDigit` and `replicate`.)

The function `printField` should satisfy the property:

```
∀n s . n >= length s ⇒ length (printField n s) == n
```

Later in the course we shall see how we can use these properties to test the correctness of a program, or even proved that such properties must always hold for a given program.

Exercise 3.1.4. Write a function `printRow :: [(Int, String)] -> String` that, given a list of pairs—the left element giving the desired length of a field and the right element its contents—formats one row in the table. For example,

```
printRow [(5, "Alice"), (6, "Allen"), (6, "female"), (6, "82000")]
```

should return the formatted row

```
"|Alice|Allen |female| 82000|"
```

(Hint: use the functions `intercalate`, `map` and `uncurry`.)

Exercise 3.1.5. Write a function `columnWidths :: Table -> [Int]` that, given a table, computes the necessary widths of all the columns. (Hint: use the functions `length`, `map`, `maximum` and `transpose`.)

Exercise 3.1.6. Write a function `printTable :: Table -> [String]` that pretty prints the whole table. (Hint: use the functions `map`, `toUpper` and `zip`.)

3.1.3 Querying

Finally we will write a few simple query operations to extract data from the tables.

Exercise 3.1.7. Write a function `select :: Field -> Field -> Table -> Table` that given a column name and a field value, selects only those rows from the table that have the given field value in the given column. For example, applying the query operation

```
select "gender" "male"
```

to the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Alice|female|
|Bob  |male  |
|Carol|female|
|Dan  |male  |
|Eve  |female|
+-----+------+
```

should result in the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Bob  |male  |
|Dan  |male  |
+-----+------+
```

If the given column is not present in the table then the table should be returned unchanged. (Hint: use the functions `(!!)`, `elemIndex`, `filter` and `maybe`.)

Exercise 3.1.8. Write a function `project :: [Field] -> Table -> Table` that projects several columns from a table. For example, applying the query operation

```
project ["last","first","salary"]
```

to the table

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Carol|Clarke|female| 50000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

should result in the table

```
+------+-----+------+
|LAST  |FIRST|SALARY|
+------+-----+------+
|Allen |Alice| 82000|
|Clarke|Carol| 50000|
|Evans |Eve  |275000|
+------+-----+------+
```

If a given column is not present in the original table it should be omitted from the resulting table. (Hint: use the functions `(!!)`, `elemIndex`, `map`, `mapMaybe`, `transpose`.)

3.1.4 Wrapping up

We can tie parsing, printing and two query operations together using:

```
exercise :: [String] -> [String]
exercise = parseTable> select "gender" "male"
                     > project ["last","first","salary"]> printTable
```

and have the program reads and write from and to standard input and standard output using:

```
main :: IO ()
main = interact (lines> exercise> unlines)
```

3.1.5 Reflection

The following questions are intended to help provoke some reflective thoughts about the exercise you just made and what you just learned. You can leave the answers in a comment at the end of your source code. Write clearly, but briefly; say neither more, nor less, than is necessary.

Question 3.1.1. Assume you were asked to implement this program in an imperative programming language such as C♯ or Java, but were not provided with as much guidance on how to structure your program, nor had made this exercise in a functional language before. How would you have structured your program?

Question 3.1.2. Now that you have made this exercise in a functional programming language, do you think that you would implement this program differently in an imperative language than if you had not? Especially think about the amount of work a particular function does, the types of the functions, the use of higher-order functions, and the use of side-effects (`System.Console.WriteLine` is a side-effecting function!)

Question 3.1.3. In Exercise 3.1.3 we mentioned that `printField` should satisfy the property ∀`n s . n >= length s` ⇒ `length (printField n s) == n`. Does it also satisfy the property ∀`n s . length (printField n s) == n`? Would it be a problem in this program if it would not?

Question 3.1.4. The property ∀`n s . n >= length s` ⇒ `length (printField n s) == n` is written in a mixture of predicate logic and Haskell. Could you express it as a mixture of predicate logic and C♯ or Java? How would you have phrased this property if you had implemented `printField` using `System.Console.WriteLine`? If I wrongly claimed that your program does not meet its specification, which of the formulations would you prefer to use to argue that you deserve a higher grade?

Question 3.1.5. In the function `printTable`, how often do you compute the required widths of the fields?

Question 3.1.6. While we guaranteed the input database contains at least one column, it is actually possible to create an "empty" table with no columns using the `project` operation (We will be nice and not test for this corner case, however.) How do you think such an empty table should be represented in both its abstract and its concrete syntax? Could someone else have a different opinion on this matter? Which of your functions would you have to modify to correctly handle this case?

3.2 Data structures for game state representation (∗∗)

In this exercise we will implement a simple game called *Butter, Cheese and Eggs* (also known as *Tic-Tac-Toe* or *Noughts-and-Crosses* on the other side of the sea). You may have played this game before, but if you're a little foggy on the rules then you can have a look at `https://en.wikipedia.org/wiki/Tic-tac-toe`.

3.2.1 Rose trees

A *rose tree* or *multi-way tree* is a tree data structure in which each node can store one value and each node can have an arbitrary number of children. Rose trees can be represented by the algebraic data type:

```
data Rose a = a :> [Rose a]
```

Note that here `:>` is a constructor written in *infix notation*. In your source code you can write it as `:>`. In Haskell, infix constructors can only consist of symbols and must start with a colon.

Exercise 3.2.1. Write functions `root :: Rose a -> a` and `children :: Rose a -> [Rose a]` that return the value stored at the root of a rose tree, respectively the children of the root of a rose tree.

Exercise 3.2.2. Write functions `size :: Rose a -> Int` and `leaves :: Rose a -> Int` that count the number of nodes in a rose tree, respectively the number of leaves (nodes without any children).

3.2.2 Game state

The *state* of a (turn-based board) game will generally consist of the player whose turn it is and what is currently on the board. The current player in a two-person game can be represented by the data type:

```
data Player = P1 | P2
```

Exercise 3.2.3. Write a function `nextPlayer :: Player -> Player` that given the player whose move it is currently, will return the player who will make a move during the next turn.

The board in *Butter, Cheese and Eggs* consists of nine fields, each either containing a cross or a circle, or being blank:

```
data Field = X | O | B
```

Exercise 3.2.4. Write a function `symbol :: Player -> Field` that gives the symbol a particular player uses. (By centuries-old tradition the first player always uses a cross.)

A *row* consists of three horizontally, vertically or diagonally adjacent fields:

```
type Row = (Field, Field, Field)
```

We can then compose the board from three horizontal rows:

```
type Board = (Row, Row, Row)
```

Exercise 3.2.5. This representation gives us easy access to the horizontal rows, but not to the vertical and diagonal ones. Write two functions `verticals :: Board -> (Row, Row, Row)` and `diagonals :: Board -> (Row, Row)` that do.

Exercise 3.2.6. Define a constant `emptyBoard :: Board` that represents the empty board.

Exercise 3.2.7. Write a function `printBoard :: Board -> String` that nicely formats a board as a string. For example, `printBoard someBoard` should return the string `"O| | \n-+-+-\n |X| \n-+-+-\n | | \n"`.

3.2.3 Game trees

A *game tree* is a rose tree where all the nodes represent game states and all the children of a node represent the valid moves than can be made from the state in the parent node.

Exercise 3.2.8. Write a function `moves :: Player -> Board -> [Board]` that, given the current player and the current state of the board, returns all possible moves that player can make expressed as a list of resulting boards. (For now, you should continue making moves, even if one of the players has already won.)

Exercise 3.2.9. Write a function `hasWinner :: Board -> Maybe Player` that, given a board, returns which player has won or `Nothing` if none of the players has won (either because the game is still in progress, or because it is a draw).

Exercise 3.2.10. Write a function `gameTree :: Player -> Board -> Rose Board` that computes the game tree. (Here you should make sure that you stop making moves once one of the players has won.)

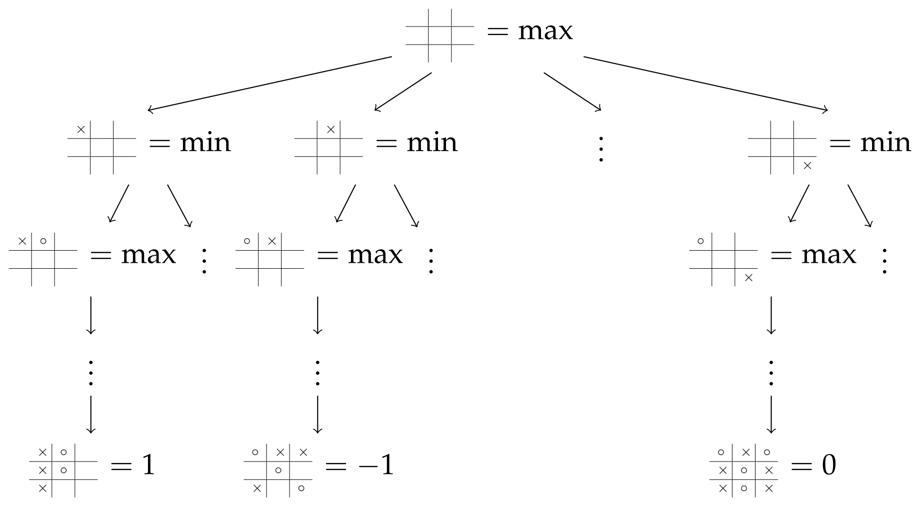
3.2.4 Game complexity

Game theorists have defined various measures of how hard a particular game is, called the *complexity* of a game. One of those measures is the *game tree complexity*, which is the number of leaves in the game tree.

Exercise 3.2.11. Define a constant `gameTreeComplexity :: Int` that computes the game tree complexity of Butter, Cheese and Eggs.

3.2.5 Minimax

We can use a game tree to implement an intelligent computer opponent (AI) for us to play against. This can be done using the *minimax* algorithm. The name of this algorithm stems from the fact that if we would assign a score to each leaf of the game tree (1 if we win, 0 if it's a draw, and -1 if we lose) then for each internal node where we make a move, we always make a move that *maximizes* our score (and take this as the score for the internal node), while for internal nodes where the opponent makes a move, we can assume they make a move that *minimizes* our score (and take this as the score for the internal node).



Exercise 3.2.12. Write a function `minimax :: Player -> Rose Board -> Rose Int` that computes the minimax tree for a given player and game tree. Here are some hints:

1. You must treat leaves of the tree (nodes that do not have any children) differently from the internal nodes of the tree (nodes that do have children).
2. The first argument of `minimax` is the Player you are calculating the minimax tree for. This argument is kept constant throughout the whole computation. It is useful to introduce a helper function `minimax'` that takes another Player argument. This is the player that is allowed to make a move, and alternates at every level of tree as you recurse through it.

If you correctly and elegantly implemented the `minimax` function then its implementation should use the `minimum` and `maximum` functions. These functions find the minimum and maximum of arbitrary lists of numbers and will thus always have to look at every element in the lists. However, we know that the lists we encounter will only contain the numbers 1, 0 or -1. Thus if `minimum` (respectively `maximum`) encounters the element -1 (respectively 1) we already know what the optimum is going to be and do not have to continue looking any further.

Exercise 3.2.13. Write lazier versions of `minimum` and `maximum` (and name them `minimum'` and `maximum'`) that stop looking for a smaller, respectively larger, number in their input list once they encounter the optimum of -1, respectively 1.

If you replace the calls to `minimum` and `maximum` with `minimum'` and `maximum'` in the `minimax` function, then the function will stop looking for an optimum once it has already found one. By "magic" of lazy evaluation the program will not only stop looking for a more optimal optimum that can never exist, it will also not bother generating whole parts of the minimax and game trees. This will make the program run several times faster.

Exercise 3.2.14. Write a function `makeMove :: Player -> Board -> Maybe Board` that makes an optimal move (if it is still possible to make a move).

3.2.6 Wrapping up

The starting framework contains a `main` function that—assuming you have implemented all of the above exercise correctly—allows you to play the game against a human or computer opponent. Of course—again assuming you have implemented all exercises correctly—you will never be able to beat the computer opponent.

3.2.7 Further research

Here are some suggestion for "further research", if you are finished early with the assignment and are feeling bored. You do not have to hand them in.

1. Generalize the game to k^n (i.e., n-dimensional Butter, Cheese and Eggs on a $k \times \cdots \times k$ board). It will be inconvenient to represent your board as nested tuples in this case. Instead, consider using an Array.
2. Use the same techniques to play other games such as Connect Four, Othello, Checkers, Chess, or Go; to solve 15 Puzzles or Rubik's Cubes; or to optimize Starcraft build orderings. For these games you are unlikely to be able to fully traverse the whole game tree to determine who wins, so you will probably want to use alpha–beta pruning and an evaluation function on a *partial game tree* instead of minimax on a complete game tree.
3. You can use the various rotational and reflective symmetries of the game board to reduce the size of the of the game tree by two orders of magnitude. Additionally, some sequences of moves will eventually result in the same game board, so it can be advantageous to represent the game as a directed acyclic graph instead of a tree. Use these techniques to speed up the AI even further.

3.3 Type classes and containers (∗∗)

In this assignment we'll present a few type classes and ask you to implement some instances of them.

3.3.1 Functors

Recall the rose tree data structure from the previous assignment:

```
data Rose α = α ▷ [Rose α]
```

Similarly to how we might want to apply a function uniformly to all elements in a list, we might also want to apply a function uniformly to all the elements in a rose tree, or any other container-like data structure for that matter. For this purpose Haskell has a `Functor` type class, exposing a single function `fmap` that generalizes the `map` function:

```
class Functor f where
  fmap :: (α -> β) -> f α -> f β
```

We see that `fmap` generalizes `map` by giving a `Functor` instance for lists:

```
instance Functor [] where
  fmap = map
```

Verify that `fmap` and `map` have the same type if we instantiate `f` to `[]`.

Exercise 3.3.1. Write a `Functor` instance for the `Rose` data type.

3.3.2 Monoids

A *monoid* is an algebraic structure over a type `m` with a single associative binary operation `(◇) :: m -> m -> m` and an identity element `mempty :: m`.

```
class Monoid m where
  mempty ::           m
  (◇)    :: m -> m -> m
```

Lists are monoids:

```
instance Monoid [] where
  mempty = []
  (◇)    = (++)
```

Verify that `(++)` is an associative operation (i.e., that `∀xs ys zs . (xs ++ ys) ++ zs == xs ++ (ys ++ zs)`) and that the empty list `[]` is indeed an identity element with respect to list concatenation `(++)` (i.e., that `∀ls . [] ++ ls == ls` and `∀ls . ls ++ [] == ls`).

Numbers also form a monoid, both under addition with 0 as the identity element, and under multiplication with 1 as the identity element (verify this). However, we are only allowed to give one instance per combination of type and type class. To overcome this limitation we create some `newtype` wrappers:

```
newtype Sum     α = Sum     {unSum     :: α}
newtype Product α = Product {unProduct :: α}
```

Now we can give two instances:

```
instance Num α => Monoid (Sum α) where
  mempty          = Sum 0
  Sum n1 ◇ Sum n2 = Sum (n1 + n2)
```

Exercise 3.3.2. Complete the second instance by writing a `Monoid` instance for numbers under multiplication.

3.3.3 Foldable

If `f` is some container-like data structure storing elements of type `m` that form a monoid, then there is a way of folding all the elements in the data structure into a single element of the monoid `m`.

```
class Functor f ⇒ Foldable f where
  fold :: Monoid m ⇒ f m -> m
```

In the case of lists:

```
instance Foldable [] where
  fold = foldr (◇) mempty
```

Exercise 3.3.3. Write a `Foldable` instance for `Rose`.

It might be the case that we have a container-like data structure storing elements of type α that do not yet form a monoid, but where we do have a function of type `α -> m` that makes them into one. In such situation it would be convenient to have a function `foldMap :: Monoid m ⇒ (α -> m) -> f α -> m` that first injects all the elements of the container into a monoid and then folds them into a single monoidal value.

Exercise 3.3.4. Add a default implementation of `foldMap` to the `Foldable` type class, expressed in terms of `fold` and `fmap`.

Exercise 3.3.5. Write functions `fsum, fproduct :: (Foldable f, Num α) ⇒ f α -> α` that compute the sum, respectively product, of all numbers in a container-like data structure.

3.4 Class instances and datastructures for a game of Poker (∗∗)

If we want to implement a poker game, we need to represent playing cards, hands and have way of ranking hands:

```
data Rank = R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | J | Q | K | A
     deriving (Bounded, Enum, Eq, Ord)
data Suit = S | H | D | C
     deriving (Bounded, Enum, Eq, Ord, Show)
data Card = Card {rank :: Rank, suit :: Suit}
type Deck = [Card]
```

3.4.1 Show

Exercise 3.4.1. Define a `Show` instance for `Rank` that shows the ranks as `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`, `"10"`, `"J"`, `"Q"`, `"K"`, `"A"` respectively.

Exercise 3.4.2. Give a `Show` instance for `Card` showing `Card {rank = R2, suit = H}` as `"2H"`.

Exercise 3.4.3. Define constants `fullDeck, piquetDeck :: Deck` that give a full 52-card deck and a 32-card Piquet deck (with cards of ranks from 7 up to and including the Ace).

3.4.2 Ord

A poker hand can be represented as:

```
newtype Hand = Hand {unHand :: [Card]}
```

and the various hand categories as:

```
data HandCategory
      = HighCard      [Rank]
      | OnePair       Rank   [Rank]
      | TwoPair       Rank   Rank   Rank
      | ThreeOfAKind  Rank   Rank   Rank
      | Straight      Rank
      | Flush         [Rank]
      | FullHouse     Rank   Rank
      | FourOfAKind   Rank   Rank
      | StraightFlush Rank
     deriving (Eq, Ord, Show)
```

If you are not familiar with the ranking of poker hands then `https://en.wikipedia.org/w/index.php?title=List_of_poker_hands&oldid=574969062` would be a good place to refer to.

The fields stored together with each category are used to distinguish between hands of the same category. For example, for a `HighCard` the field of type `[Rank]` contains all five cards in the hand, sorted from high to low. In the case of a `TwoPair` the first `Rank` is that of the high pair, the second `Rank` that of the low pair, and the third `Rank` that of the kicker (the card that isn't part of any of the two pairs).

Convince yourself that the derived `Ord` instance correctly ranks poker hands represented as a `HandCategory`.

We are now going to write a function that converts hands of type `Hand` into hands of type `HandCategory`. First we'll need a few helper functions:

Exercise 3.4.4. Write a function `sameSuits :: Hand -> Bool` that returns `True` if all cards in a `Hand` are of the same suit.

Exercise 3.4.5. Write a function `isStraight :: [Rank] -> Maybe Rank` that return a `Just` with the highest ranked card, if the `Hand` is a straight (or a straight flush). Note that the Ace can count both as the highest and as the lowest ranked card in a straight!

Exercise 3.4.6. Write a function `ranks :: Hand -> [Rank]` that converts a `Hand` into a list of `Ranks`, ordered from high to low.

Exercise 3.4.7. Write a function `order :: Hand -> [(Int, Rank)]` that converts a `Hand` into a list of `Ranks` paired with their multiplicity, order from high to low using a lexicographical ordering. For example, the hand `["7H", "7D", "QS", "7S", "QH"]` should be ordered as `[(3, R7), (2, Q)]`.

Exercise 3.4.8. Finally, write a function `handCategory :: Hand -> HandCategory` that converts a `Hand` into a `HandCategory`.

Exercise 3.4.9. Using `handCategory`, write an `Ord` instance for `Hand`.

3.4.3 Combinatorics

Exercise 3.4.10. Write a function `combs :: Int -> [a] -> [[a]]` that returns all the combinations that can be formed by taking *n* elements from a list.

Exercise 3.4.11. Write a function `allHands :: Deck -> [Hand]` that returns all combinations of 5-card hands than can be taken from a given deck of cards

3.4.4 Data.Set

The `Ord` class on `Hand` induces an equivalence relation on poker hands. This can be useful when using functions or data structures that require an `Ord` instance on the data they are working with.

One example of such a data structure is `Data.Set`: a data structure that can only store unique elements. Uniqueness is determined by the equivalence relation induced by an `Ord` instance.

Exercise 3.4.12. Write a function `distinctHands :: Deck -> Set Hand` that constructs a maximal set of distinct hands from deck. (Hints: You will need the `empty` and `insert` functions from `Data.Set`. Use `foldl'` instead of `foldr` to avoid a stack overflow when applying this function to large decks.)

3.4.5 Questions

Question 3.4.1. Do numbers[1] form a monoid under subtraction? If so, give the identity element. Do numbers form a monoid under division?

Does `Bool` form a monoid under conjunction `(&&)`? Does `Bool` form a monoid under the biconditional `(==)`?

Question 3.4.2. Sheldon wants to implement Rock-paper-scissors-lizard-Spock in Haskell. He defined a data type:

```
data Gesture = Rock | Paper | Scissors | Lizard | Spock
```

and now wants to define an `Ord` instance for this data type that specifies which of two gestures beats the other.

Explain why this is not a good idea. The answer can be found by carefully reading the documentation of `Data.Ord` or imagining what happens if you sort a list of gestures using such an ordering.

3.5 Monads for a gambling game (***)

In this assignment we'll ask you to implement a probability monad and an instrumented state monad.

3.5.1 A Game of Chance

Here's a game I like to play: I toss a coin six times and count the number of heads I see, then I roll a dice; if the number of eyes on the dice is greater than or equal to the number of heads I counted then I win, else I lose. As I'm somewhat of a sore loser, I'd like to know my chances of winning beforehand, though.

There are three ways to compute this probability:

1. Use a pen, paper (or, if you prefer, chalk and a blackboard) and some basic discrete probability theory to calculate the probability directly.
2. Draw or compute the complete decision tree of the game and count the number of wins and losses.
3. Write a computer program that simulates the game to approximate the probability.

[1]If you're now asking yourself: "But what kind of numbers do you mean exactly, Sir?" then please consider both various Haskell types having a `Num` instance (`Integer`, `Rational`, `Float`, ...) as well as various mathematical classes of numbers ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}\backslash\{0\}$, ...).

As we're computer scientists, we'll leave the first option to the mathematicians and focus on the second and third possibilities. In fact, using monads, we'll see how both can be done at the same time.

The Gambling Monad

Modeling a coin and a dice in Haskell shouldn't pose much difficulty for you anymore:

```
data Coin    = H | T
data Dice    = D1 | D2 | D3 | D4 | D5 | D6
data Outcome = Win | Lose
```

The tossing of a `Coin` and rolling of a `Dice` is given by the monadic interface `MonadGamble`:

```
class Monad m => MonadGamble m where
  toss :: m Coin
  roll :: m Dice
```

Exercise 3.5.1. Write a function `game :: MonadGamble m => m Outcome` that implements the game above. Read the description of the game very carefully: it is easy to make an off-by-one error; furthermore, as tossing and rolling are side-effects the order in which you perform them matters.

Simulation

Simulating probabilistic events requires a (pseudo)random number generator. Haskell has one available in the `System.Random` library. Random number generators need to have access to a piece of state called the *seed*, as such the random number generator runs in a monad, the `IO` monad to be exact.

Exercise 3.5.2. Give `Random` instances for `Coin` and `Dice`.

Exercise 3.5.3. Give a `MonadGamble` instance for the `IO` monad.

Exercise 3.5.4. Write a function `simulate :: IO Outcome -> Integer -> IO Rational` that runs a game of chance (given as the first parameter, not necessarily the `game` implemented in Exercise 3.5.1) n times ($n > 0$, the second parameter) and returns the fraction of games won.

You can now approximate to probability of winning using `simulate game 10000`. Would you care to take a guess what the exact probability of winning is?

Decision trees

One drawback of simulation is that the answer is only approximate. We can obtain an exact answer using decision trees. Decision trees of probabilistic games can be modeled as:

```
data DecisionTree a = Result a | Decision [DecisionTree a]
```

In the leaves we store the result and in each branch we can take one of several possibilities. As we don't store the probabilities of each decision, we'll have to assume they are uniformly distributed (i.e., each possibility has an equally great possibility of being taken). Fortunately for us, both fair coins and fair dice produce a uniform distribution.

Exercise 3.5.5. Give a `Monad` instance for `DecisionTree`. (Hint: Use the types of (>>=) and `return` for guidance: it's the most straightforward, type-correct definition that isn't an infinite loop. The definition will somewhat resemble that of the `Monad` instance for `Eval`, as asked for in the last exercise in Hutton's monad material. In both cases they're so-called *free monads*.)

Exercise 3.5.6. Give a `MonadGamble` instance for `DecisionTree`.

Exercise 3.5.7. Write a function `probabilityOfWinning :: DecisionTree Outcome -> Rational` that, given a decision tree, computes the probability of winning.

You can find the exact probability of winning using `probabilityOfWinning game`. Was your earlier guess correct? If you know a bit of probability theory, you can double check the correctness by doing the pen-and-paper calculation suggested above.

Note that we used the same implementation of `game` to obtain both an approximate and an exact answer.

3.5.2 Instrumented State Monad

In Hutton's monad material we encountered the state monad (which he called `ST`, as it are technically state transformers—functions mapping one state to another—that form a monad). We'll now give a presentation of the state monad that is closer to how they are found in Haskell's `Control.Monad.State` library.

A state monad is monad with additional monadic operations `get` and `put`:

```
class Monad m => MonadState m s | m -> s where
  get    ::          m s
  put    :: s      -> m ()
  modify :: (s -> s) -> m s
```

(The "`| m -> s`" part of this class is called a *functional dependency*. You can ignore this. If you want to know exactly what it does, then you should follow the *Advanced Functional Programming* course during your Master's. The short answer is that it helps the compiler figure out which particular state monad instance it needs to use for a given type.)

Apart from the usual three monad laws, state monads should also satisfy:

```
put s₁ >> put s₂              == put s₂
put s  >> get                 == put s >> return s
get    >>= put                ==          return ()
get    >>= (\s -> get >>= k s) == get   >>= (\s -> k s s)
```

Check to see if you understand what these four laws say and if they make sense.

Exercise 3.5.8. Give default implementations of `get` and `put` in terms of `modify`, and a default implementation of `modify` in terms of `get` and `put`.

Instrumentation

We are now going to define our own, slightly modified state monad that, besides keeping track of a piece of state, has also been instrumented to count the number of `(>>=)`, `return`, `get` and `put` operations that have been performed during a monadic computation.

The counts are given by the type:

```
data Counts = Counts {
  binds   :: Int,
  returns :: Int,
  gets    :: Int,
  puts    :: Int
}
```

Exercise 3.5.9. As a convenience, give a `Monoid` instance for `Count` that sums the counts pairwise. Define constants `oneBind, oneReturn, oneGet, onePut :: Counts` that represent a count of one `(>>=)`, `return`, `get` and `put` operation, respectively.

Our state transformer is now given by:

```
newtype State' s a = State' {runState' :: (s, Counts) -> (a, s, Counts)}
```

Note that our `State'` corresponds to Hutton's `ST`, but that it has been parameterized over the type of state `s` (for which Hutton used the type synonym `State`). Additionally, we keep track of the `Counts` as an internal piece of state that is not exposed through the `get` and `put` interface.

Exercise 3.5.10. Give `Monad` and `MonadState` instances for `State'` that count the number of `(>>=)`, `return`, `get` and `put` operations.

Tree labeling

Here is another tree data type:

```
data Tree a = Branch (Tree a) a (Tree a) | Leaf
```

This is a binary tree that stores values on the internal nodes only.

Exercise 3.5.11. Write a function `label :: MonadState m Int => Tree a -> m (Tree (Int, a))` that labels a tree with integers increasingly, using a depth-first in-order traversal.

Exercise 3.5.12. Write a function `run :: State' s a -> s -> (a, Counts)` that runs a state monadic computation in the instrumented state monad, given some initial state of type `s`, and returns the computed value and the number of operations counted.

For example, the expression

```
let tree = Branch (Branch Leaf "B" Leaf) "A" Leaf
in  run (label tree) 42
```

should evaluate to

```
(Branch (Branch Leaf (42, "B") Leaf) (43, "A") Leaf
, Counts {binds = 10, returns = 5, gets = 4, puts = 2})
```

3.5.3 Further reading

If you want a bit more practice with implementing functions in the `IO` monad, then try implementing the Mastermind game.

If you want to know more about the probability monad then have look at "Probabilistic Functional Programming in Haskell", Martin Erwig and Steve Kollmansberger, *Journal of Functional Programming*, Vol. 16, No. 1, pp. 21–34, 2006.

3.6 Database (∗∗)

This assignment deals with (a subset of) SQL, the *structural query language* for dealing with data in relational databases. As often, Wikipedia has a good summary on the history and the look of the language.

In current database systems, SQL is used not only to describe queries, but also to describe the format of tables and their contents. In this assignment, we implement a very simple database system entirely in Haskell, and develop a small SQL parser together with an interpreter so that we can describe data for that database and run queries. It is *not* an assignment that teaches you how to interface to a real database system.

For this task, you are supposed to parser combinators in order to write the parser. You can use any parser combinator library you like.

Structure

This assignment consists of several parts: You have to devise the abstract syntax of a fragment of the SQL grammar, to write a parser using the parser combinators. Furthermore, you have to implement the database system, and you have to write functions that interpret the SQL abstract syntax in terms of actions on the database system.

It is important to realize that there is not just one order in which you can approach this task. The representation of the database itself in terms of Haskell datatypes is given. Hence, you can start implementing database operations in Haskell and later write the parser and the semantics functions. Or, you can start with the parser, then write the semantic functions and implement the database in the end.

You can even decide to mix the approaches, and first implement some language constructs completely, and then move on to the next. Do whatever you feel works best, and if you get stuck somewhere, try if you can make progress in another area.

SQL grammar

The fragment of SQL we consider is given by the following grammar:

```
program      ::= tablecommand* query
tablecommand ::= create | insert
create       ::= CREATE TABLE name ( names? )
insert       ::= INSERT INTO name insertion
insertion    ::= VALUES ( entries? )
               | query
query        ::= SELECT names FROM names wherepart?
wherepart    ::= WHERE expression
expression   ::= expression AND expression
               | expression OR expression
               | NOT expression
               | ( expression )
               | operand operator operand
names        ::= name , names | name
entries      ::= entry , entries | entry
entry        ::= string | number
operand      ::= name | entry
operator     ::= < | > | =
```

Terminals are written in typewriter font, nonterminals in italics. Between any two symbols (terminals or nonterminal) in the above rules, arbitrary amounts of whitespace are allowed.

In addition, there are the following nonterminals for the lexical syntax – *no* whitespace is allowed anywhere in these rules:

name	::= *letter alphanum**
string	::= ' *char** '
number	::= *digit digit**
alphanum	::= *letter* | *digit*
letter	::= any letter
digit	::= any digit
char	::= any character except the single quote '

Exercise 3.6.1. Define Haskell datatypes to describe the abstract syntax of SQL. First apply the general strategy to come up with a first version, but then reflect and consider if you cannot optimize a bit by introducing lists and occurrences of `Maybe`. Call the datatype for the starting nonterminal *program* `Program`.

Hint: if you want to split the syntax into smaller chunks, then omit *wherepart* and *expression* for the beginning.

Exercise 3.6.2. The fact that whitespace can occur almost everywhere within an SQL statement makes the definition of parsers slightly trickier than usual. Nevertheless, we can define our parser in a single step, without the need for a separate lexical analyzer. The idea is to make most of the parsers consume additional spaces at the end of the input.

Define a parser

```
spaces :: Parser String
```

that greedily parses as much whitespace as possible (use the `isSpace` function from module `Data.Char`). You can use the declaration

```
import Data.Char
```

in the beginning of your program in order to import a module.

Then, use this parser to define parsers

```
keyword :: String -> Parser String
parens  :: Parser a -> Parser a
commas  :: Parser a -> Parser [a]
```

These are variants of `token`, `parenthesised` and `listOf` from `ParseLib`, but allow whitespace after each token.

Example:

```
parens (keyword "SELECT") "(  SELECT   ) "
```

should successfully parse the string "SELECT", but

```
parens (keyword "SELECT") "(SEL ECT)"
```

and

```
parens (keyowrd "SELECT") "  (SELECT)"
```

should fail (i.e., drop the spaces in while error-correcting, because no spaces are allowed in the middle of a keyword, or in the beginning).

Exercise 3.6.3 (medium)**.** The grammar, as given, is ambiguous (and left-recursive) for expressions. Remove the ambiguity and left recursion by assuming that NOT, AND, and OR have the usual priorities. Allow parentheses to be used to explicitly group expressions. Note that you do not have to reflect this in the abstract syntax (i.e., the Haskell datatypes), but you have to use the transformed grammar in order to define the parser.

Exercise 3.6.4 (medium)**.** Define a parser

```
parseProgram :: Parser Program
```

that can parse an SQL program. Define all the other parsers for the other nonterminals. Make use of the combinators defined above, and define more if required. The guideline should always be that every parser consumes additional spaces in the end, but not at the beginning. Only the parser for the complete program parseProgram should also allow spaces at the beginning.

You can test your parser on some simple SQL programs, for instance

```
INSERT INTO foo VALUES (2,3,4)
SELECT name,location FROM addresses
```

Note that you can, for debugging purposes, also test individual parsers you define. In fact, try to verify every parser you define on a few examples.

Database implementation

We are going to implement our database directly in Haskell in a very straightforward way, with a focus on simplicity rather than efficiency. We model the entire database as a finite map. In order to get finite maps into your program, you should add the following import statements to the module header of your solution:

```
import qualified Data.Map as M
import Data.Map (Map (..))
```

Now, the database maps table names to tables:

```
type DB    = Map TName Table
type TName = String
```

A table contains the names of its columns, plus a list of rows that are the entries in the table.

```
data Table   = Table Columns [Row]
  deriving Show
type Columns = [Name]
type Name    = String
type Row     = [Entry]
```

As entries, we allow either strings or integers. We use the `deriving` construct to derive functions not only to show entries, but also to compare them.

```
data Entry = String String | Int Int
  deriving (Show, Eq, Ord)
```

(Note that we are introducing *constructors* `String` and `Int`, each parameterized with one argument of the indicated type.)

A simple table that associates course abbreviations and years with the name of the lecturers can be represented as follows:

```
exampleTable :: Table
exampleTable =
  Table
     ["course",          "year",   "lecturer"]
    [[String "INFOAFP",  Int 2007, String "Andres Loeh"],
     [String "INFOAFP",  Int 2006, String "Bastiaan Heeren"],
     [String "INFOFPLC", Int 2008, String "Andres Loeh"],
     [String "INFOSWE",  Int 2008, String "Jurriaan Hage"]]
```

Operations on the database can be modelled using a state monad:

```
type DBM = State DB
```

For this to work, you have to

```
import Control.Monad.State
```

This module contains the definition we discussed in the course:

```
newtype State s a = State (s -> (a, s))
```

and defines functions

```
runState :: State s a -> s -> (a, s)
put      :: s -> State s ()
get      :: State s s
```

as discussed. In addition, it also defines a function

```
modify :: (s -> s) -> State s ()
modify f =
  do
    s <- get
    put (f s)
```

that modifies the state according to the function given.

Exercise 3.6.5. Define a function

```
printTable :: Table -> String
```

that turns a table into a readable string. Print the column headers, a line of horizontal dashes, and then all the rows. The column entries should be aligned, and sufficient room should be reserved for each column to hold the widest entry that occurs in that column.

Example:

```
putStrLn (printTable exampleTable)
```

should return something like the following output:

```
course     year lecturer
----------------------------------
'INFOAFP'  2007 'Andres Loeh'
'INFOAFP'  2006 'Bastiaan Heeren'
'INFOFPLC' 2008 'Andres Loeh'
'INFOSWE'  2008 'Jurriaan Hage'
```

Exercise 3.6.6. Define a function

```
createTable :: TName -> Columns -> DBM ()
```

that, given the name of a table and column names, adds a new empty table with these column names.

Exercise 3.6.7. Define a function

```
insertInto :: TName -> Row -> DBM ()
```

that inserts a new row into a table. The function should check that the number of entries in the row matches the number of columns in the table and do nothing if the numbers don't match.

Exercise 3.6.8 (medium)**.** Define functions

```
select  :: Table -> RowProperty -> Table
project :: Table -> Columns -> Table
pair    :: Table -> Table -> Table
```

where

```
type RowProperty = Row -> Bool
```

The function `select` should keep only the rows from a table that have the given property.

The function `project` should keep only the columns from the table that are given, in that order. If columns are given that don't exist in the table, then those should be ignored.

The function `pair` should compute that cartesian product of two tables. The resulting table has all the columns of the first table, followed by all the columns from the second table. The rows are all combinations of rows from the first and rows from the second table. In particular, if the first table has c_1 columns and r_1 rows, and the second table has c_2 columns and r_2 rows, then the resulting table has $c_1 + c_2$ columns and $r_1 \cdot r_2$ rows.

Exercise 3.6.9 (difficult)**.** Define a function

```
iExpression :: Expression -> [Name] -> RowProperty
```

that – assuming that `Expression` is the datatype used to represent nonterminal *expression* – interprets an expression as a row property. The additional argument of type `[Name]` indicates the column names of the table the property is operating on.

To understand how this works, let's look at the grammar for expressions. An expression is essentially a conjunction or disjunction of conditions, where each condition is some form of comparison between operands. Operands can either be constants or (column) names.

Assuming you have defined

```
data Operand = Entry Entry | Name Name
  deriving Show
```

you can define a helper function

```
iOperand :: Operand -> [Name] -> (Row -> Entry)
iOperand (Entry e) ns r = e
iOperand (Name n)  ns r = r !! (fromJust (findIndex (==n) ns))
```

that interprets an operand correctly as either a constant value or looking up the appropriately named column from the given row. The function `iOperand` uses `fromJust` from the `Data.Maybe` module and hence will fail if an "unknown" column name is mentioned, but we will leave this for a bonus exercise to fix.

Using `iOperand`, you can now define `iExpression`.

Exercise 3.6.10 (medium)**.** Define a function

```
iQuery :: Query -> DBM Table
```

that – assuming `Query` represents the nonterminal *query* – interprets a query as an operation on the database that returns a new table. The strategy is as follows: lookup all the tables mentioned in the `FROM` part and compute their product (using `pair`). Then use the columns of the product table to turn the `WHERE` part into a row property using `iExpression`. Use `select` to apply the row property to all the rows of the product table, and finally `project` to keep the columns specified in the `SELECT` part of the query.

Exercise 3.6.11. Write a function

```
iTableCommand :: TableCommand -> DBM ()
```

that – assuming that `TableCommand` is the datatype for the nonterminal *tablecommand* – interprets an insert or create command as a database operation. Since an insert command can contain a query, you will have to use `iQuery` to define `iTableCommand`.

Exercise 3.6.12. Write a function

```
iProgram :: Program -> DBM Table
```

that interprets an entire SQL program and returns the table delivered by the final query.

Exercise 3.6.13. Write a

```
main :: IO ()
```

that uses the function

```
getArgs :: IO [String]
```

from the module `System.Environment` in order to check the command line arguments of the program. The argument should be interpreted as a filename, the file read using

```
readFile :: String -> IO String
```

and the contents of the file should be interpreted as a program using `parseProgram` and `iProgram`. After that, the program should print the resulting table using `printTable`.

As a final example, here is a possible input file:

```
CREATE TABLE courses (course, year, lecturer)
INSERT INTO courses VALUES ('INFOAFP',  2007, 'Andres Loeh')
INSERT INTO courses VALUES ('INFOAFP',  2006, 'Bastiaan Heeren')
INSERT INTO courses VALUES ('INFOFPLC', 2008, 'Andres Loeh')
INSERT INTO courses VALUES ('INFOSWE',  2008, 'Jurriaan Hage')
CREATE TABLE topics (subject, topic)
INSERT INTO topics VALUES ('INFOAFP', 'monad transformers')
INSERT INTO topics VALUES ('INFOFPLC', 'Haskell')
INSERT INTO topics VALUES ('INFOFPLC', 'parser combinators')
INSERT INTO topics VALUES ('INFOSWE', 'version management')
INSERT INTO topics VALUES ('INFOSWE', 'deployment')
SELECT lecturer, topic FROM courses, topics WHERE course = subject
```

and the output:

```
lecturer          topic
---------------------------------------
'Jurriaan Hage'   'deployment'
'Jurriaan Hage'   'version management'
'Andres Loeh'     'parser combinators'
'Andres Loeh'     'Haskell'
'Bastiaan Heeren' 'monad transformers'
'Andres Loeh'     'monad transformers'
```

Extra exercise for LC-only students

Exercise 3.6.14 (medium)**.** Define the algebra and fold function for your abstract syntax. Reexpress all the interpretation functions as a fold.

Bonus exercises

Exercise 3.6.15 (bonus)**.** Write a little interactive loop that reads SQL table commands or queries from the command line and interprets them. For table commands, just update the state. For queries, print the result. Update your `main` function to use the interactive loop if no filename is specified as a command line argument.

Exercise 3.6.16 (bonus, medium)**.** Add the possibility to save the current database to a file and read it from a file. This requires you to devise a good storage format for the database and be able to read that format again. An option is to use the standard `Read` and `Show` functions Haskell provides.

Exercise 3.6.17 (bonus, variable)**.** Make the program more robust by checking against all sorts of incorrect inputs (such as mentioning column names that don't exist) and giving meaningful errors in such a case.

Exercise 3.6.18 (bonus, variable)**.** Add more features of SQL to the query language. An easy extension is to allow renaming of columns with `AS` in a `SELECT` statement, or to allow selecting all columns using `ALL`. Look for SQL descriptions to get more ideas for additional commands.

3.7 TurtleGraphics ($**$)

The topic of this task is to write a simple interpreter for a simple version of the Logo programming language. More information about Logo can for instance be found in the Wikipedia article at

```
http://en.wikipedia.org/wiki/Logo_(programming_language)
```

The focus of this assignment is the use of IO in Haskell. Parts of the assignment deal with reading a file and drawing in a window.

Logo

The Logo language is a very simple language to control a turtle (that happens to carry a pen around). The turtle begins in one particular position (say, in the middle of a window), facing in one particular direction.

You can now give commands to the turtle, of the following form:

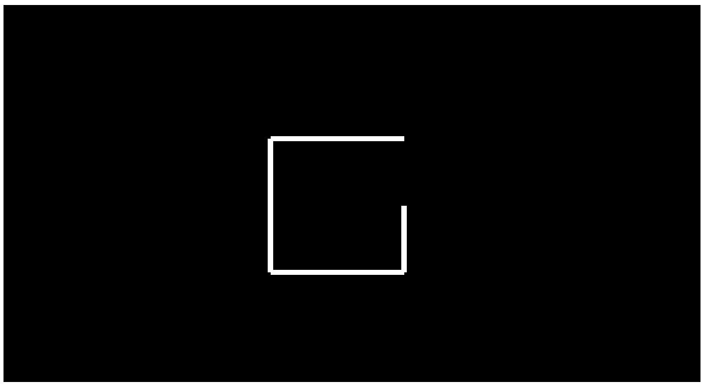
Command	Effect
`forward <num>`	The turtle steps forward by `num` steps. If the turtle is currently drawing, then a line is drawn along the turtle's path.
`back <num>`	The turtle steps back by `num` steps. If the turtle is currently drawing, then a line is drawn along the turtle's path.
`right <angle>`	The turtle turns right (i.e., clockwise) by `angle` degrees.
`left <angle>`	The turtle turns left (i.e., counterclockwise) by `angle` degrees.
`penup`	Stop drawing.
`pendown`	Start drawing.

Here is an example:

```
forward 50
right 90
```

```
forward 50
right 90
penup
forward 25
pendown
forward 25
right 90
forward 50
```

The result should look approximately as follows:



The task

The program should read a Logo program from disk, open a window and paint the results of the program in the window. Finally, the program should wait for a keypress before the window is closed again.

For drawing the resulting picture, we are using the SOE graphics library, an extremely simple (and not very powerful) graphics interface originating from the Haskell book called "The Haskell School of Expression" by Paul Hudak – hence the name SOE. There are several implementations of that library. The most easiest to install should be within the HGL library (which is available via `cabal install` from Hackage, but should be installed on the Linux lab machines already).

Depending on the library you choose, you have to import a module. If you want to use the HGL variant, simply place the line

```
import Graphics.SOE
```

in your module header.

The functions you need will be mentioned in the exercises. Note nevertheless that you can use the GHCi commands `:browse` to browse all definitions in a module, and `:info` and `:type` to get information and types of identifiers.

Steps

Exercise 3.7.1. Write a datatype `Command` that represents a single Logo command. I.e., define one constructor per command.

Exercise 3.7.2. Write a parser that turns a string into a list of commands:

```
parseLogo :: String -> [Command]
```

It is possible to do this using parser combinators, but the Logo language is so simple that this is not really needed. You can use `lines` to split text into lines, `words` to split a line into multiple words at blanks, and then `read` where appropriate to convert strings into numbers.

The function may just fail if the string contains an illegal Logo program.

Exercise 3.7.3. Now, write a function that takes a filename, reads the file and then parses it:

```
getLogo :: FilePath -> IO [Command]
```

Note that the following type synonym is predefined in the prelude:

```
type FilePath = String
```

Exercise 3.7.4. Define a type that represents the state of the turtle. Depending on your preference, this can be a type synonym, thus

```
type TurtleState = ...
```

or a datatype

```
data TurtleState = ...
```

The state has three components: the current position, the current angle where the turtle is facing, and whether the turtle is currently drawing or not.

Exercise 3.7.5. Define the initial state of the turtle. Let's code the window size as a constant for the moment, for instance

```
windowSize :: Size
windowSize = (600, 600)
```

The type synonym `Size` is defined by the SOE library as follows:

```
type Size = (Int, Int)
```

The turtle should initially be in the middle of the window, face up and be drawing.

```
initialState :: TurtleState
```

Exercise 3.7.6. Write a function that executes a single command and transforms the state. There are different ways of sophistication in which this can be achieved. The simplest approach is probably to directly draw in the window using the

```
drawInWindow :: Window -> Graphic -> IO ()
```

function. You then define

```
processCommand :: Window -> Command -> TurtleState -> IO TurtleState
```

that receives the window, the command, and the original state as arguments.

An alternative that is somewhat nicer is not to draw immediately, but to accumulate a `Graphic` that can be drawn later. This has the advantage that `processCommand` does not use `IO` and does not need the current window as a parameter.

```
processCommand :: Command -> TurtleState -> (Graphic, TurtleState)
```

Now, if you are already familiar with the state monad, this is just an instance of it. So yet another option is to use the type:

```
processCommand :: Command -> State TurtleState Graphic
```

Choose whatever you like best or find easiest. Use the following functions from the SOE library

```
line      :: Point -> Point -> Graphic
withColor :: Color -> Graphic -> Graphic
```

Note that you will have to calculate the new position from the old position using `sin` and `cos`, and that you will have to convert angles, because the `left` and `right` commands are parameterized by degrees (between 0 and 360) whereas `sin` and `cos` expect radians (between 0 and 2π). The constant π is available as `pi` in Haskell.

Exercise 3.7.7. Try to define a function that processes multiple commands. The type of this function depends on the type that you have chosen for `processCommand`, but it will be something like

```
processCommands :: ... [Command] ... -> ...
```

i.e., there should be a list of commands among the inputs.

If you do not draw directly into the window, but assemble values of type `Graphic`, you may find the function

```
overGraphics :: [Graphic] -> Graphic
```

useful that simply combines different graphics (in this case, lines) into a single graphic.

Exercise 3.7.8. Write a function that takes a list of commands, opens a window, processes the commands given the initial state, waits for a keypress, then closes the window. Here is a template for the function:

```
runLogo :: [Command] -> IO ()
runLogo cmds = runGraphics $
  do
    w <- openWindow "Logo" (600,600)
     ... processCommands ... cmds ... initialState ...
     ...
    getKey w
    closeWindow w
```

How to invoke `processCommands` depends on the type. If `processCommands` does not draw directly, you have to extract the resulting `Graphic` and draw it using `drawInWindow`.

Exercise 3.7.9. Now, combine everything with the parser. Write

```
runFile :: FilePath -> IO ()
```

that reads the file using `getLogo` and subsequently invokes `runLogo`.

Exercise 3.7.10. We are done. A final option is to create a proper `main` function that uses the command line argument as file name.

```
main = do
        args <- getArgs
        runFile (head args)
```

This will fail if there are no command line arguments. Note that `getArgs` is not defined in the prelude, but requires you to

```
import System.Environment
```

at the top of your module.

Bonus exercises

There are lots of extensions possible for our little language. However, keep in minds that the possibilities of the SOE library are quite limited, and many more advanced graphical features may require switching to a different graphics library.

Exercise 3.7.11 (bonus)**.** Add a possibility to draw in different colors.

Exercise 3.7.12 (bonus, medium)**.** Create a possibility to save the current state and return to it later, in a nested, stack-like way, using commands `push` and `pop`.

Exercise 3.7.13 (bonus, medium)**.** Add loop constructs such that for example

```
repeat 4
  forward 50
  right 90
end
```

draws a square. The main difficulty here is that parsing becomes less straight-forward. Note that you can also only implement everything but the parsing, starting from extending the `Command` datatype. Logo programs are rather simple to write in Haskell.

Exercise 3.7.14 (bonus, medium)**.** Taking the hint from the previous exercise, devise a nice embedded domain-specific language in Haskell in order to create values of type `[Command]`.

Exercise 3.7.15 (bonus, difficult)**.** Extend the Logo language with variables.

Any other ideas are also fine. Use your imagination.

3.8 Stereograms (∗∗)

The topic of this assignment are *single-image random dot stereograms* (SIRDS for short). SIRDS are images that contain a "hidden" three-dimensional structure. There is lots of information about SIRDS on the web, including techniques on how to see the hidden images and many examples. See for instance Wikipedia at

```
http://en.wikipedia.org/wiki/SIRDS
```

for a relatively detailed explanation including lots of links to more information. Figure **??** shows an example SIRDS in black and white that contains a chessboard pattern similar to that in Figure 3.1. Note that stereograms are sensitive to the resolution, so you have to print the stereogram without scaling, or – if you want to watch it on screen – have to view the file at 100% and with any visual modifications (such as antialiasing) turned off.

There are many variations on the stereogram idea. There are images consisting of just random dots, but also artful pictures where even the two-dimensional variant is a joy to look at. There are images which encode several three-dimensional pictures when looked at in different ways, and there are animated stereograms as well.

In this task, we will concentrate on a really simple algorithm that produces pictures composed of pixels of more or less random colors – in other words, the two-dimensional pictures just look like (somewhat repetitive) noise.

The task consists of the following components:

- Writing a routine to output images.
- Defining a data structure that can maintain 'links' between pixels.
- Defining an algorithm that can compute a SIRDS from a depthmap.

Additionally, an algorithm that can (partially) reconstruct the depthmap from a SIRDS to help debugging.

Images

The first part of the task is to write code that can write images to disk, so that you can view them using an image viewer. We are going to use the PPM format to write images, because it is one of the simplest image formats available. Lots of converters exist than can automatically translate PPM files into more common graphics formats (such as for instance PNG) for you.

If for some reason, you cannot find such a conversion program or viewer for PPM, I recommend that you try to implement a writer for BMP instead, which is another really simple image format, but not quite as simple as PPM.

In essence, each PPM file consists of a header followed by the image data. The header identifies the file as a PPM file, contains the resolution and the color depth of the image. The data is an encoding of each pixel in the image, line by line.

In Haskell, we represent a single pixel by its RGB color, i.e., its red, green and blue color components, where each component ranges from 0 to 255:

```
type Color = Int
data RGB   = RGB Color Color Color
```

Note that we use RGB both as name of the datatype and as name of the single constructor. Such double use is allowed by Haskell and quite common. Nevertheless, the constructor RGB is a function, whereas the datatype RGB is a type. Both exist in parallel.

An image is then a list (rows) of a list (columns) of RGB colors where all rows have equally many columns:

```
type Image = [[RGB]]
```

Exercise 3.8.1. Write functions

```
validColor :: Color -> Bool
validRGB   :: RGB   -> Bool
```

that check if a given color or RGB value is valid, i.e., if all colors involved are integers in the correct range.

Exercise 3.8.2 (medium). Write a function

```
validImage :: Image -> Maybe (Int, Int)
```

that tests if an image is valid and, if it is, returns the resolution of the picture as a pair of the x-resolution and the y-resolution. An image is valid if all its pixels are valid *and* if all the rows have equally many columns. All these properties should be checked, and `Nothing` should be returned for all invalid images. Example:

```
*Main> validImage []
Just (0, 0)
*Main> let p = RGB 0 0 0; r = [p, p, p] in validImage [r, r, r, r]
Just (3, 4)
*Main> let p = RGB 0 0 0; r = [p, p, p] in validImage [r, r, r, [p]]
Nothing
*Main> validImage [[RGB 1 2 257]]
Nothing
```

Exercise 3.8.3. The next step is to generate the PPM header for an image. For this, you have to define a function

```
ppmHeader :: (Int, Int) -> String
```

The header consists of four components. The components are separated by a space, the whole header is terminated by a newline. The first component is always the string `"P6"` that is an identification for the PPM file format. The second and third components are the x- and y-resolution of the image, respectively – they are therefore passed as parameter of `ppmHeader`. The final component gives the maximum color value we are using. For the purposes of this assignment, all our images will use colors from `0` to `255`, so the final component is the constant string `"255"`. Hint: Recall that you can use `show` to turn an integer into its string representation. Example:

```
*Main> ppmHeader (1024, 768)
"P6 1024 768 255\n"
```

Exercise 3.8.4. Of course, we must also encode the image data. Therefore, define functions

```
encodeRGB :: RGB   -> String
ppmData   :: Image -> String
```

The former encodes a single pixel, the latter encodes a whole image by encoding all the pixels in order and concatenating the encodings (without any spaces or newlines). How is a pixel encoded? For each RGB-value, we generate a string of length 3 with one character for each component. We encode each component by using the character with the corresponding ASCII code. For this, there is a function `chr :: Int -> Char` (defined in the module `Data.Char`. Example:

```
*Main> encodeRGB (RGB 65 66 67)
"ABC"
*Main> ppmData [[RGB 65 66 67,RGB 68 69 70],[RGB 71 72 73,RGB 74 75 76]]
"ABCDEFGHIJKL"
```

Note that some ASCII code correspond to non-printable characters that will be escaped if shown by the interpreter:

```
*Main> encodeRGB (RGB 0 1 2)
"\NUL\SOH\STX"
```

Exercise 3.8.5 (medium)**.** We can now define a function that writes a PPM image to a file:

```
writePPM :: FilePath -> Image -> IO ()
```

Here, `FilePath` is an abbreviation for a string

```
type FilePath = String
```

You will have to use most of the functions defined so far. The function should test the given image for validity. If the image isn't valid, it should write an error message to the screen. If it is valid, however, it should write the PPM encoding of the image (the header concatenated with the data) to the specified file. To write a file, use the function

```
writeBinaryFile :: FilePath -> String -> IO ()
```

that is predefined in the program skeleton.

Exercise 3.8.6. Verify that your PPM writer works. In the skeleton, there are predefined images `chess`, `gradient` and `circular` that you can write to files of your choice using `writePPM`. Do so, then use a converter/viewer to display the files on screen and check that they look as expected (i.e., as in Figures 3.1–3.3.

Links

The basic idea of the SIRDS generation algorithm is that one pixel in the imaginary three-dimensional target landscape is mapped to two pixels – one for each eye – in the stereogram. To make it possible for the eyes to match up these corresponding pixels, such pixels should have the same color. The SIRDS generation algorithm determines which pixels should be 'linked' in this way. Let us call the distance between two linked points the 'length' of that link. A point in the three-dimensional space that is close to

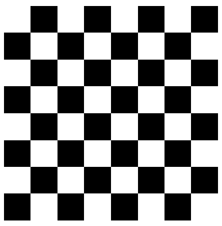


Figure 3.1: chessboard

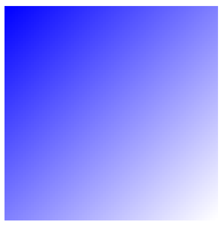


Figure 3.2: linear gradient

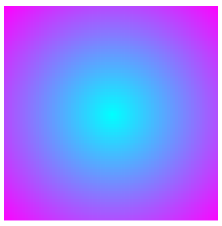


Figure 3.3: circular gradient

the observer leads to a short link, whereas a point that is further away leads to a long link. The idea is shown in Figure 3.4.

A pixel can be linked to the left and to the right at the same time. The left and right eye then perceive that pixel as different points in the three-dimensional space.

Things in the foreground tend to be perceived slightly larger than things in the background – therefore, points in the three-dimensional space that are further away are sometimes obscured (shadowed) by other points. In practice, this means that links can collide. Several three-dimensional points might be linked to the same two-dimensional positions. In this case, shorter links (corresponding to closer points) win over longer links.

Exercise 3.8.7. Let us introduce types to represent the links we are interested in. A point (we are only interested in x-coordinates here, therefore points are represented as integers for now) can either be linked with another point, or unlinked:

```
data Link = Linked Int Int | Unlinked Int
```

We maintain the invariant that for the `Linked` constructor, the first point is always smaller than (i.e., to the left of) the second point. We also call the first point the *left* point, and the second the *right* point. Here is a function that tests the invariant:

```
validLink :: Link -> Bool
validLink (Linked x y) = x < y
validLink (Unlinked _) = True
```

Because – as argued above – shorter links shadow longer links, define an operator

```
(>%>) :: Link -> Link -> Bool
```

(read as 'better'). A link is better than another if it is strictly shorter than the other. A `Linked` link is always better than an `Unlinked` point. You can choose any result for comparing two `Unlinked` points – it does not matter for the stereogram algorithm.

Exercise 3.8.8 (medium). Next, we will implement a data structure that maintains all the links we discover. We use so-called *finite maps* for this purpose. Finite maps are defined in the module `Data.Map`. For instance, the type

```
Map Int Char
```

is a finite map from integers to characters, and can contain a finite number of associations between integers and characters. The first type is also called the *key* type, the second the *element* type. Keys of type integer can be used to look up characters.

Study the Haddock (a documentation generation tool for Haskell) documentation of the `Data.Map` module, available from

`http://haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html`

For the upcoming exercises, you will need to use several functions from this module, so you should try to understand what the offered methods do and test them in GHCi on example inputs.

The module has been imported into the skeleton program you are working on using the statement

```
import qualified Data.Map as M
```

This makes all functions from the module available for use in your program and also in GHCi, but you have to prefix the names with an `M` – for instance, you have to use `M.lookup` rather than `lookup`. The reason we are doing this is that some functions on lists have the same names as their finite map counterparts, and we want to be able to distinguish between the different versions.

Check at least `empty`, `lookup`, `insert` and `delete`.

Try to understand the types of these functions. Make up example expressions such as

```
*Main> M.lookup 2 (M.insert 2 5 (M.insert 2 3 M.empty)) :: Maybe Int
*Main> M.lookup 3 (M.delete 3 (M.insert 3 6 M.empty)) :: Maybe Int
```

until you feel comfortable working with finite maps. In particular, `M.lookup` is difficult to understand, because it has a rather general type

```
M.lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

The result type can be delivered in an arbitrary monad, but for our purposes, it will suffice to think of `m` as `Maybe` here. To force this instantiation, use explicit type annotations for the result type like shown in the examples above.

Exercise 3.8.9 (medium). Here is how we store all the links:

```
type Links = Map (Int, Dir) Int
data Dir   = L | R
```

The name `Links` abbreviates a finite map. The keys of the map are pairs of an integer (an x-coordinate) and a direction (`L` for 'left' or `R` for 'right'). As explained initially, any x-coordinate can be linked to a point on the left and to a point on the right. If `x` is a specific point, then `(x, R)` is the key for the point to the right, and `(x, L)` is the key for the point to the left.

An invariant is attached to our use of this data structure: if `(x, R)` is mapped to a point `x'`, then `(x', L)` should be mapped back to `x`, and vice versa.

Write a function

```
add :: Link -> Links -> Links
```

that adds a single link to the finite map such that the invariant is maintained (i.e., *two* entries have to be added to the finite map). You may assume that no other links with the same left or right point are in the `Links` data structure at the time of the insertion. Adding an `Unlinked` point should leave the finite map unchanged.

Using the given definition of

```
noLinks :: Links
noLinks = M.empty
```

add a few links to an empty finite map in GHCi and verify that your function is working. Next, write a function

```
del :: Link -> Links -> Links
```

that removes a link from the finite map. Removing an `Unlinked` point should again leave the finite map unchanged. For a `Linked` link, you may assume that the link exists in the finite map. You have to maintain the invariant, though, i.e., you will have to remove *two* entries from the finite map. Test this function as well.

Exercise 3.8.10 (medium)**.** Write a function

```
query :: Link -> Dir -> Links -> Link
```

to query the data structure of links. A call of the form `query (Linked l r) L` or `query (Unlinked r) L` should check if right point `r` is already linked with another left point. I.e., if `Linked l' r` is already in the structure, then `Linked l' r` should be returned. If there is no link in the structure that has `r` as right point, `Unlinked r` is returned. Similarly, the calls `query (Linked l r) R` or `query (Unlinked l) R` check if the left point `l` is already linked with another right point.

Exercise 3.8.11 (difficult)**.** Write a function

```
link :: Link -> Links -> Links
```

that adds a link `Linked l r` to the structure, but only if it's 'better' (according to >%>) than both the links that are returned by querying for `Linked l r` in both directions. If the new link is indeed better, then first the old links should be removed using `del`, and finally the new link should be added using `add`. If the new link is not better or if the new link is `Unlinked`, then the `Links` data structure should be returned without modification.

Stereogram generation

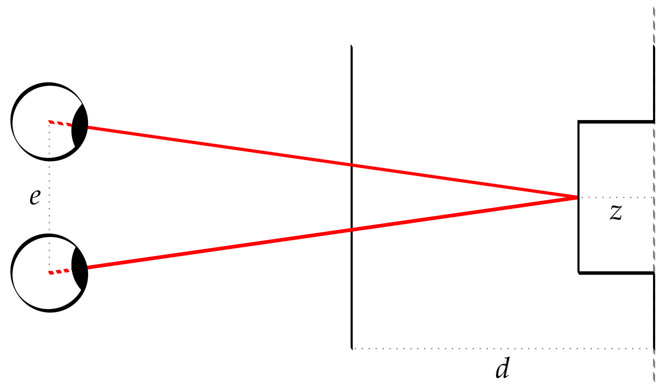


Figure 3.4: The idea of SIRDS generation

We are now at a point where we can actually implement the generation of a SIRDS relatively easily. The input to the algorithm is a *heightmap*:

```
type HeightMap = [[Height]]
type Height    = Double
```

This is a list (lines) of lists (columns) of z-coordinates of the same size than the intended resulting image. Each of the heights is supposed to be between `0.0` and `1.0` for optimal viewing results.

Exercise 3.8.12. Write a function

```
separation :: Double -> Int
```

that computes the separation of the two linked points from a given height value. Look at Figure 3.4. The input of this function is the height z of the 3D-image point from the base plane of the 3D-image (grey dashed line in the picture). The output is supposed to be the distance of the two points where the rays from the eyes to that point cross the plane of the 2D-image.

The separation can be calculated using the following formula:

$$\text{separation}\, z = \frac{e(d-z)}{2d-z}$$

Transcribe this formula to Haskell to define the `separation` function. The parameters `e` (the distance between the eyes) and `d` (the distance between the eyes and the 2D-image, which is equal to the distance between the 2D-image and the imaginary base plane of the 3D-image) are given in the skeleton.

For our purposes, the final result should be an integer, because we want a distance in pixels. Use the `round` function to convert a fractional number into an integral number.

Exercise 3.8.13 (difficult)**.** Write a function that processes a single line of the stereogram and produces the according links:

```
sirdsLine :: [Height] -> Links
```

The length of the original list determines the width of the line, lets call it `width`. Start with a structure with no links. For each x-position `x`, you have to compute the separation `s` corresponding to the height at this position, and then link `x - (s 'div' 2)` with `x - (s 'div' 2) + s` using the function `link`, but *only* if both points are on the line, i.e., at least `0` and at most `width - 1`. The final structure of links is then returned. Hint: It is probably easiest to define an additional help function that has extra arguments for the state you have to maintain, i.e., the width, the current x-position and the current `Links` data structure.

Exercise 3.8.14. At this point, you are done, because the remaining functions are all given. To complete the job, we make use of the function

```
assign :: Int -> Links -> IO [RGB]
```

that assigns random colors to a single line (the width of the line is the first argument) while respecting that linked points should get the same color. Because `assign` makes use of random numbers, and random numbers are a side-effect, its result is an `IO` type. Assign internally makes use of the function

```
findRightMost :: Links -> Int -> Int
```

that, given an x-coordinate `x`, finds the rightmost x-coordinate that is chain-linked with `x`, and that therefore has to be of the same color.

Finally, there is a function

```
sirds :: HeightMap -> IO Image
```

that performs the complete conversion from a heightmap into an SIRDS. For each line of the input, `sirdsLine` is called to compute all the links. Then `assign` is called to assign colors that respect the links. Study the code of these functions and try to understand it.

You can then try to run the main program. Note that the program will run *significantly* faster if you compile it with optimizations using `ghc -O --make SIRDS.hs`.

Decoding stereograms

I'm aware that not everyone can see stereograms, and that non-working stereogram algorithms can be difficult to debug. I have therefore provided an algorithm that can recover the heightmap from a SIRDS. This is a lossy operation.

The function is called

```
decode :: Image -> Image
```

and takes an encoded (SIRDS) image as input and produces the decoded variant as a heightmap in graytones. Areas that cannot be decoded are printed in red. Red areas will typically appear at the left and right of the whole image, and to the left and right of high points, because those points shadow lower points that are directly adjacent. Here is an example, as performed by the default `main` routine. The `doubleChess` function can be transformed into a graytone image using the `hightmap` function, resulting in the image shown in Figure 3.5. If encoded as a stereogram via `sirds` and decoded back using `decode`, you will end up with something like Figure 3.6.

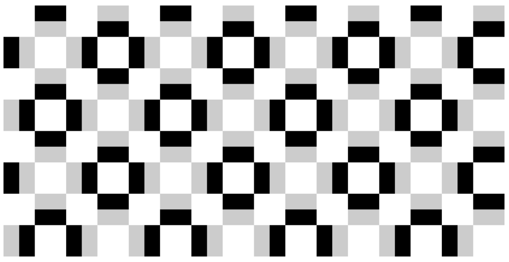


Figure 3.5: double chessboard pattern

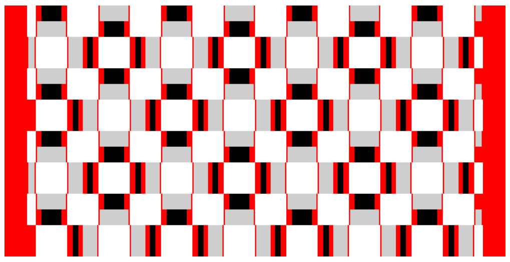


Figure 3.6: en- and decoded pattern

Bonus exercises

Exercise 3.8.15 (bonus, more tricky than difficult)**.** Add a PPM reader, and a converter from full-color PPMs into heightmaps. Then you can generate your own heightmaps with a graphics program of your choice.

Exercise 3.8.16 (bonus)**.** Write a command-line interface that reads the input heightmap and the output PPM as command-line arguments, and then performs the requested conversion. Look at the `System.Environment` library for the necessary functions to access command line arguments.

Exercise 3.8.17 (bonus, difficult)**.** Try to implement an SIRDS algorithm that takes a pattern as input, and does not just use random dots.

Exercise 3.8.18 (bonus, difficult)**.** Try to make the SIRDS decoding algorithm more intelligent, so that it can handle SIRDS that have been generated by other means.

3.9 Arrow (∗∗)

The goal of this assignment is to implement a little domain-specific programming language. Programs comprise instructions for a little spaceship called *Arrow* that flies around in bounded two-dimensional space. The space is not empty, but inhabited with various flying objects such as asteroids, lambdas and debris. By interpreting programs, we can let Arrow fly through space and perform certain tasks such as finding a way through an asteroid field and cleaning up debris.

Credits

This assignment is inspired by the *Kara* programming system:

```
http://www.swisseduc.ch/compscience/karatojava/kara/
```

and in particular by Frank Huch's paper "Learning Programming with Erlang" that appeared in the proceedings of the 2007 ACP SIGPLAN workshop on Erlang.

Alex and Happy

For this task, you are supposed to use the Alex lexer generator and the Happy parser generator. These are available from

```
http://haskell.org/alex/

http://haskell.org/happy/
```

but they are also part of the *Haskell Platform*, so if you have that installed, you already have both Alex and Happy. In particular, you should be able to invoke them from the command line on the lab machines.

The Arrow programming language

The concrete syntax of the Arrow language is given by the following grammar with start symbol `Program`:

```
Program -> Rule*
Rule    -> Ident -> Cmds .
Cmds    -> ε | Cmd (, Cmd)*
Cmd     -> go | take | mark | nothing
         | turn Dir
         | case Dir of Alts end
         | Ident
Dir     -> left | right | front
Alts    -> ε | Alt (; Alt)*
Alt     -> Pat -> Cmds
Pat     -> Empty | Lambda | Debris | Asteroid | Boundary | _
```

A program is a sequence of rules. Think of rules as procedures. A name is bound to a sequence of commands. Rules are terminated by a period.

Commands are separated by commas. There is a fixed number of commands. These are instructions for *Arrow*. Informally, `go` means "move in the current direction if possible", `take` means "pick up whatever is here", `mark` means "leave a lambda in the current spot", `nothing` means "do nothing", `turn` takes a direction and causes Arrow to turn left or right. The `case` command takes a direction and performs a sensor reading in that direction. Depending on what is sensed, different actions may be taken. Finally, another rule can be invoked by naming it.

In a `case` construct, multiple alternatives can be provided (separated by semicolons) that map patterns to rules. Patterns correspond to the things that can be located in a certain position, and there is a catch-all pattern called `_`.

Note that unlike in Haskell, `case` expressions are terminated by an `end` keyword.

The lexical syntax of a program is described as follows: the program text consists of a (possibly space-separated) sequence of tokens.

```
Token -> -> | . | , | go | take | mark | nothing | turn | case | of | end
         | left | right | front | ;
         | Empty | Lambda | Debris | Asteroid | Boundary | _
         | Ident
Ident -> (Letter | Digit | + | -)+
```

A token is either symbolic, a command keyword, a pattern keyword, or an identifier. It is implicitly understood that an `Ident` must not be any of the keyword tokens and must not be directly followed by another character that could occur in an identifier.

Furthermore, comments may occur in programs between tokens. These are introduced by `--` and extend to the end of the line.

Exercise 3.9.1. Define a suitable abstract syntax for the Arrow language. Call the type corresponding to a whole program `Program`.

Exercise 3.9.2. Write a parser for the language using parser combinators.

Exercise 3.9.3 (medium)**.** Define functions to perform analysis of the program that (next to possibly required additional information) performs the following sanity checks on a given program:

- There are no calls to undefined rules (rules may be used before they are defined though).
- There is a rule named `start`.
- No rule is defined twice.
- There is no possibility for pattern match failure, i.e., all `case` expressions must either contain a catch-all pattern `_` or contain cases for all five other options.

Define a function

```
check :: Program -> Bool
```

that combines all of the above checks and returns true iff a program is sane.

An interpreter for Arrow programs

Arrow lives on a rectangular board that we call "space" and represent using a finite map (from module `Data.Map`):

```
type Space    = Map Pos Contents
type Size     = Int
type Pos      = (Int,Int)
data Contents = Empty | Lambda | Debris | Asteroid | Boundary
```

We assume that there always is a rectangular area of positions with non-negative row- and column-coordinates contained in the finite map, including position `(0,0)`. We leave the size of the space open though, and functions can use `findMax` to find the maximum key and hence the maximum position in a given space.

We define an input format for spaces where contents are represented by single characters:

contents	character
empty	`.`
lambda	`\`
debris	`%`
asteroid	`O`
boundary	`#`

We specify the format by example:

```
(7,7)
........
....%...
..%%%%..
....%%%.
...%%%..
....%.%%
....%%%%
........
```

The first line contains the maximum valid row-column-coordinate for the board. Here, we thus have a space with 8 rows and 8 columns. The rows are then specified line by line, starting with row 0 end ending with row 7. The example space contains a field of debris, but otherwise just empty space.

A parser for the input format can be written as follows (assuming the `uu-parsinglib` library – code can probably be adapted to work with other parser combinator libraries):

```
parenthesised :: Parser a -> Parser a
parenthesised = pPacked (pSym '(') (pSym ')')
natural :: Parser Int
natural = read <$> pList1 (pSym (isDigit,"digit",'0'))
parseSpace :: Parser Space
parseSpace =
  do
    — read dimensions of the 'Space'
    (mr,mc) <- parenthesised ((,) <$> natural <* pSym ',' <*> natural) <* spaces
    — read 'mr + 1' rows of 'mc + 1' characters
    css <- replicateM (mr + 1) (replicateM (mc + 1) contents)
    — convert from a list of lists to a finite map representation
    return $ fromList $ concat $
      zipWith (\r cs -> zipWith (\c d -> ((r,c),d)) [0..] cs) [0..] css
```

The function `replicateM` is defined in `Control.Monad`, so that module has to be imported. We still need the parser `contents` that parses a single character and maps it to the appropriate constructor of type `Contents`:

```
    contents :: Parser Contents
    contents = pAny (\(f,c) -> f <$ pSym c) contentsTable <* spaces
    contentsTable :: [(Contents,Char)]
    contentsTable =
      [(Empty,'.'),(Lambda,'\\'),(Debris,'%'),(Asteroid,'O'),(Boundary,'#')]
```

Exercise 3.9.4. Write a printer for `Space` that produces the output format just shown.

Exercise 3.9.5. Assuming that `Ident` is the Haskell type representing an identifier, and `Commands` represents a sequence of commands, we represent a program as an environment during execution:

```
    type Environment = Map Ident Commands
```

Write a function

```
    toEnvironment :: String -> Environment
```

that first lexes and then parses a string, checks the resulting `Program` using `check`, and, assuming the check succeeds, translates the `Program` into an environment.

Exercise 3.9.6 (medium)**.** During the execution of a program, we have to maintain state. The state contains the current space, the position of Arrow, its heading, and a stack of commands.

```
type Stack      = Commands
data ArrowState = ArrowState Space Pos Heading Stack
```

Implement a function that performs a single execution step:

```
step :: Environment -> ArrowState -> Step
```

where `Step` encodes the possible results of one execution step:

```
data Step = Done Space Pos Heading
          | Ok   ArrowState
          | Fail String
```

The function implements the following semantics. The top item on the command stack is analyzed:

- On `go`, Arrow moves forward one step using its current heading, as long as the target field is empty or contains a lambda or debris. Otherwise, it stays where it is.
- On `take`, Arrow picks up lambda or debris, leaving an empty space at its current position.
- On `mark`, Arrow places a lambda at its current position regardless of what was there before (debris is removed).
- On `nothing`, nothing changes.
- On `turn`, Arrow changes its heading by 90 degrees to the left or right as indicated. Turning `forward` is possible, but has no effect.
- On a `case`, Arrow makes a sensor reading. Depending on the direction specified as an argument to `case`, Arrow will take a look at the position that – according to its current heading – is to the front, left, or right. The pattern of each alternative is then analyzed in turn until one matching alternative is found. The instructions on the right hand side are then prepended to the command stack and execution continues. If no alternative matches, execution fails. An alternative matches if the pattern corresponds to the contents. Positions that are not stored in the finite map are implicitly assumed to contain `Boundary`. A catch-all pattern matches always.
- On a rule call, the code stored with that rule in the environment is prepended to the command stack. If the rule is not defined, execution fails.
- If the command stack is empty, a `Done` result is produced.

Exercise 3.9.7. Rules can be recursive. Note how recursion affects the size of the command stack during execution. Does it matter whether the recursive call is in the middle of a command sequence or at the very end of the command sequence? Include your observations as a comment or in a separate file.

Exercise 3.9.8. Write a driver

```
interactive :: Environment -> ArrowState -> IO ()
```

that – given an environment and an initial state – runs the program interactively. In every step, the driver should print at least the board and ask for some form of user confirmation. After getting the user input, the driver should invoke the next step and continue from the beginning. The driver should recognize abnormal and successful terminations of the reduction and treat them sensibly.

Example: Remove debris

The following example program removes all debris in a connected component of the space. So, for instance, running this program on the example space shown above with the ship starting in any position filled with debris should ultimately clear all the debris in the space, then stop.

```
start -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right,
         s2.

s2    -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right,
         s3.


s3    -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                   go, turn right, turn right;
             _      -> nothing
            end,
            turn right,
            s4.

s4    -> take,
            case front of
              Debris -> go, start, turn right, turn right,
                        go, turn right, turn right;
              _      -> nothing
            end,
            turn right.
```

Adding natural numbers

Here is another example program that adds two natural numbers:

```
start       -> turn right, go, turn left, firstArg.

turnAround  -> turn right, turn right.

return      -> case front of
                 Boundary  ->  nothing;
                 _         ->  go, return
               end.

firstArg    -> case left of
                 Lambda  ->  go, firstArg, mark, go;
                 _       ->  turnAround, return, turn left,
                             go, go, turn left,
                             secondArg
               end.

secondArg   -> case left of
                 Lambda  ->  go, secondArg, mark, go;
                 _       ->  turnAround, return, turn left,
                             go, turn left
               end.
```

The program expects its input as rows of lambdas, as in the following example:

```
                          (4,14)
```

```
\\\\\..........
...............
\\\\\\\........
...............
...............
```

The first number here is 5, the second 7.

If you start arrow facing east in the upper left corner of the space (i.e., at position `(0,0)`), then the result of adding the two numbers is written below the two inputs:

```
(4,14)
\\\\\..........
...............
\\\\\\\........
...............
\\\\\\\\\\\\...
```

Bonus exercises

Exercise 3.9.9. Write a proper main program that lets you read in a space and a program from a file, specify a start position and heading, and runs the interactive driver.

Exercise 3.9.10 (easy to difficult)**.** Extend the interactive driver:

- Print extra information such as the current contents of the stack (set a useful cutoff limit).
- Allow the user to request multiple steps being performed without asking for confirmation.
- Allow going back in the execution.
- Add a full debugger that allows setting breakpoints.

Exercise 3.9.11. Write a non-interactive driver

```
batch :: Environment -> ArrowState -> (Space,Pos,Heading)
```

Exercise 3.9.12. Write lots of interesting programs in the Arrow language.

Exercise 3.9.13 (medium)**.** Extend the language such that you can abstract over code blocks, and rules can have parameters. The example program for removing debris could be much simplified with this extension.

Exercise 3.9.14 (difficult)**.** Add a graphical driver.

4 Larger tasks: game programming

4.1 MasterMind (*)

The purpose of this assignment is to reimplement the Mastermind game – see the Wikipedia article at

```
http://en.wikipedia.org/wiki/Mastermind_%28board_game%29
```

for more information about the game.

The game is sufficiently small to fit into a single Haskell module, distributed in a file `MasterMind.hs`. The file contains a skeleton program: it runs and typechecks, but doesn't do anything really useful yet.

Several functions in the program are missing or only partially implemented. The places where you have to modify or add stuff are marked using calls to the dummy function `tODO`. In the final program, you should have replaced all the occurrences of `tODO` with meaningful code.

Overview

Mastermind is a game for two players, called the codemaker and the codebreaker. The codemaker's role is played by the computer in our case. The codemaker devises a code made up of four positions, each position being one of six colors (represented by the numbers 1 to 6). The codebreaker (played by the human user of your program) must guess the code in as few turns as possible. After each guess, two scores are determined for the guess. The *black* score says how many positions of your code match the solution. Once the black score is 4, the codebreaker has determined the correct code and the game is over. The *white* score indicates that a position of the codebreaker's code used a color (number) contained in the solution, but in the wrong position. It is easy to see that the sum of black and white score never exceeds 4, the number of positions in the code.

Here are two protocols of possible games:

code	guess	score
3 4 6 6	1 1 2 2	0 black, 0 white
3 4 6 6	3 3 4 4	1 black, 1 white
3 4 6 6	3 5 3 6	2 black, 0 white
3 4 6 6	3 4 6 6	4 black, 0 white

code	guess	score
5 1 1 4	1 2 3 4	1 black, 1 white
5 1 1 4	1 3 5 6	0 black, 2 white
5 1 1 4	5 2 1 5	2 black, 0 white
5 1 1 4	5 2 4 1	1 black, 2 white
5 1 1 4	5 4 1 1	2 black, 2 white
5 1 1 4	5 1 1 4	4 black, 0 white

And here is how the game looks being played using a Haskell program:

```
? 2 2 3 3
0 black, 1 white
? 4 4 5 5
0 black, 1 white
? 5 2 1 6
1 black, 3 white
? 5 1 6 2
4 black, 0 white
Congratulations.
```

The game interactively prompts the user to type in a guess using a `?` symbol. The user types in a whitespace-separated sequence of four numbers between 1 and 6, and the game responds with the score. If a black score of 4 is reached, the game stops – otherwise, it asks for another guess.

A bottom-up approach

To solve the problem of implementing the game, we will implement several functions in an incremental fashion. If all functions are implemented correctly, you should be able to play the game yourself.

Exercise 4.1.1. Read the skeleton. Compile the skeleton and run it. See what the program does (and more importantly, what it doesn't do). Find all the positions marked `tODO`. Apart from the definition of `tODO` itself, there are seven such positions.

Exercise 4.1.2 (medium)**.** Let us start with the function `black`. Both a guess and a solution are represented by a list of integers `[Int]`. We introduce abbreviations `Guess` and `Solution` for this type, so that our type signatures can be more descriptive. Currently, the function `black` always returns `0`. Redefine the function so that it computes the `black` score correctly. Test the function on inputs of your choice – for example, on the inputs from the game protocols given above:

```
*Main> black [5,1,1,4] [1,2,3,4]
```

```
1
*Main> black [3,4,6,6] [3,5,3,6]
2
```

Exercise 4.1.3 (difficult)**.** Note that computing the white score is much more difficult than computing the black score. Don't feel forced to do the exercises in this order – if you get stuck, try the rest first and come back to this exercise later.

Write function `white` which, given the solution and a guess, computes the white score. Again, test your function on examples of your choice, for instance

```
*Main> white [5,1,1,4] [1,2,3,4]
1
*Main> white [3,4,6,6] [3,5,3,6]
0
```

Exercise 4.1.4. Extend the definition of `check` such that the third component tests if the guess was all-correct and the game could be finished. Again, test your function on inputs of your choice:

```
*Main> check [5,1,6,2] [5,2,1,6]
(1,3,False)
*Main> check [5,1,6,2] [5,1,6,2]
(4,0,True)
```

Exercise 4.1.5. The function `report` takes the result of `check` and assembles a piece of text that can be presented to the user. It should indicate the score the user has achived, and also give a message in the case the game was won. The output doesn't have to match the examples here exactly, but should be approximately as follows:

```
*Main> report (check [5,1,6,2] [5,2,1,6])
"1 black, 3 white"
*Main> report (check [5,1,6,2] [5,1,6,2])
"4 black, 0 white\nCongratulations."
```

Note that you can use the function `putStrLn` to print a string on the screen, thereby interpreting escape sequences such as the newline `\n`:

```
*Main> putStrLn it
4 black, 0 white
Congratulations.
```

Exercise 4.1.6 (medium)**.** The function `input` has type `IO Guess` – it should read a whitespace-separated sequence of numbers from the screen, turn it into a list of integers, and return that list as a guess. Currently, it reads a line into the string `l`, but doesn't transform the string, and instead returns the empty list. Fix this problem by using the prelude function `words` (check what it does again) and the given function `readInt` (read the documentation and test it in the interpreter to see how it works). Finally, test your function:

```
*Main> input
? 3 4 5 5
[3,4,5,5]
*Main> input
? 7 hello
[7,-1]
```

Note that the function currently does not check its input to be valid. The user can place non-numeric inputs, enter colors that are outside the range from 1 to 6, or specify less or more than four colors.

Exercise 4.1.7 (medium)**.** Now we can assemble all our work in the function `loop`. The function should call `check` on the input, use `report` to generate output for the player, and it should repeat the loop unless the given guess was correct. Once you've implemented that function, the game is playable. The `main` function produces a random code, and then calls `loop` with that solution. The function `generateSolution` that generates the random code is given in the skeleton, so you can test the function `loop` by running `main` from the interpreter or by compiling your program (see in the beginning) into an executable and running that.

Bonus exercises

Exercise 4.1.8 (bonus)**.** Implement the function `valid` that, given a guess as produced by `input`, decides if the guess is valid according to the rules of the game. I.e., the guess should contain `width` numbers, and the numbers should be between `1` and `colors` – we store the game parameters in constants so that they're easier to parameterize over at a later point in the development. Then change `input` such that it calls `valid` on the guess before returning it. If the user input isn't valid, it should complain and ask for new input rather than returning the invalid guess.

Exercise 4.1.9 (medium, bonus)**.** Improve the game experience: Count the number of turns the player requires and print it in the end. Allow the user to give up, and in that case, print what the correct solution would have been.

Exercise 4.1.10 (difficult, bonus)**.** Try to implement an algorithm for playing the game. Two such algorithms are given on the Wikipedia page. Modify the game so that the computer plays against itself, and prints the protocol.

4.2 Lambdarinth (***)

4.2.1 The game

In the beginning, the game to implement will be described. The game, called "Lambdarinth", is – apart from minor variations – an instance of the board game "Ricochet Robots" (designed by Alex Randolph and published by various companies).

The game is played on a rectangular board (in the original, always 16x16) with an unlimited number of players (although, for practical reasons, at least one player is good to have). Between some of the squares and surrounding the whole board are walls. A number of Lambdas (in the original, always 4) of different colors are placed randomly on the board.

A problem is then posed to all the players, namely to reach a certain square with a specific Lambda. In order to solve the problem, any Lambda may be moved both horizontally and vertically, but only until it is blocked either by a wall or another Lambda. Players should try to get to the target square in as few moves as possible. However, players can just look at the board, not perform any actual moves (yet). Once they know a solution, they can shout out the number of moves their solution has. From then on, there is a timeout of typically 60 seconds. Within this time, other players can announce that they have also found solutions by shouting other numbers. Players who have already announced a solution can also improve their solutions by announcing lower numbers.

After the timeout passes, the winner is determined: The player with the lowest number who voiced the solution before any other players with the same number first gets the chance to demonstrate the solution on the board. If the player cannot show a valid solution (or the solution has too many moves), the player with the next-best solution gets a chance. If none of the players can demonstrate a solution, the game remains without a winner.

4.2.2 The task

The overall description of the task is simple. You should implement both a server and a client for the game just described. All code should be written in Haskell. You may use libraries and tools that are available on Hackage.

Client and server should communicate with each other via a network socket, on a configurable port, making use of a textual protocol that is specified in Section 4.2.3. The protocol also makes the basic structure of the game explicit.

In Section 4.2.4, additional requirements for the server and client are specified.

Finally, in Section 4.2.5, a number of ideas for extensions and improvements are given.

You should try to implement a number of improvements on the initial requirements. Let your imagination go wild. There are basically no limitations, except that your client and server should still provide a mode in which they will be interchangeable with other clients and servers that implement the game protocol.

Demo server

I will try to run a demo server on the host `shell.students.cs.uu.nl`, port `7890`. You can connect to this server in order to test your own clients. The server should, in principle, implement the game and the game protocol as specified in this document, with the exception of board generation and the rejection of too simple problems as described in Section 4.2.4. If you think it does not, please let me know. I may update the server during the block for fixes, improvements or extensions. I will then tell you so. The server may also be temporarily unavailable if problems show up, so do not rely on its presence too much. Nevertheless, if you find the server down, let me know so that I can try to bring it up again.

Since the game protocol is textual, you can also use `telnet` to connect to the server and enter commands textually. From a Unix machine within the student network, the command

```
telnet shell.students.cs.uu.nl 7890
```

should connect to the server.

4.2.3 The protocol

Messages sent between the servers and the clients all have a common form. Each message is a single line terminated with a period immediately before the end of the line. The first word in the message identifies the kind of message. We call this word the *command*.

Since the protocol might be extended, or because incorrect clients might connect to the server, the server is required to be very robust. Whenever a message (i.e., input line) is received that is not in the required format, the message should just be discarded and not further affect the state of the server.

Clients should also be tolerant and just ignore messages they do not understand.

Clients and server make use of a different set of commands. Here is a list of all server and client commands that should be understood.

Phases of the game

We can distinguish three phases the server can be in: **Before**, **In**, or **After** a game.

Before a game the board and the positions of the Lambdas are already known, but not the problem (i.e., the target square). Players can confirm their interest in playing the following game in this phase.

After a certain time, or when all players around have confirmed, the game starts. The confirmed players participate, the others just watch. While **In** the game, players who are participating can make a bid by providing a number of moves in which they believe they can solve the problem. All bids are collected by the server, but bids with fewer moves are better than bids with more moves. If several bids have the same number of moves, the one first given counts. If several bids are provided by the same player, only the lowest counts. The game takes some amount of time. After the first bid, the remaining time is set to 60 seconds regardless of what it has been before. When this timeout has passed, the game ends.

After the game the server tries to determine a winner. In the order of the bids received, it asks the players who submitted the bids to submit a solution. There is a timeout for that as well. If a player who is asked manages to submit a correct solution in time, the server announces a winner and moves to the **Before** phase of the next game. If no player can submit a correct solution, the server announces that there have been no winners and also advances to the **Before** phase of the next game.

Client commands

`Confirm` *Player*.

The client confirms that *Player* wants to participate in the upcoming game. The server answers with either a `NameTaken` or an `AcceptedPlayer` accepted for the same *Player* if it is in the **Before** phase. In other phases, the server will just ignore a `Confirm` command. Multiple `Confirm` commands in the **Before** phase of a game by a single client are possible, so a player can choose a new name as long as the game has not started.

`Bid` *Moves*.

The client states that it can provide a solution to the given problem in a maximum of *Moves* moves. The server ignores this command if it is not in the **In** phase. In the **In** phase, if the bid is currently the best bid of the player in question, the server reacts with a `ReceivedBid` command.

`Done.`

The client states that it has no interest anymore in the current game and would not mind for it to end. The server ignores this command if it is not in the **In** phase. If, while in the **In** phase, all confirmed players send a `Done` command, the server can accelerate the game and immediately switch to the **After** phase.

`Solution` *Solution*.

The client submits a solution *Solution* to the current problem. The server ignores this command if it is not in the **After** phase and has sent a `RequestSolution` command for that player before. The server also ignores the command if the solution submitted is illegal or has too many moves. If the solution was ok, the server answers with a `Winner` command.

Server commands

`Game` *Board*.

The server sends the board and the positions of the Lambdas *Board* for the next game. The server sends this command to all clients when it starts a **Before** phase, or to a new client when it connects and the server is in the **Before** or **In** phase.

`NameTaken` *Player*.

The server rejects a `Confirm` command previously sent to the server. The `NameTaken` command indicates that the chosen name *Player* is already in use by *another* player. It should not be sent in any other situation. The server sends this command as a possible reply to a `Confirm` command only to the client that sent the `Confirm` command.

`AcceptedPlayer` *PlayerId Player*.

The server announces that a player *Player* will join the upcoming game. The *PlayerId* associated with the player is uniquely determined by the client. The *PlayerId* can be used by clients to identify if the player is a new player or has just chosen a new name. The server sends this command as a possible reaction to a `Confirm` command to all connected clients.

`PlayerGone` *Player*.

The server announces to all connected clients that a player *Player* has left the game. The server uses this command if the connection to the client of a confirmed player has been lost, regardless of the phase the server is in.

`Problem` *Players Prob Timeout*.

The server announces to all connected clients a problem *Prob* on the current board. It also gives the *Players* that will play this game (i.e., that have confirmed before the game started and not left in the meantime). The *Timeout* in seconds indicates how much time there is at most to come up with a bid. After sending this command, the server is in the **In** phase.

`ReceivedBid` *Player Moves Timeout*.

The server announces to all connected clients that it has received a bid by player *Player*, to provide a solution in at most *Moves* moves. The remaining time in the game in seconds is indicated by *Timeout*. The server sends this command as a reaction to a `Bid` command, but only if the server is in the **In** phase, and if the bid it has received is the best bid the player in question has made in the current game so far.

`RequestSolution` *Player Moves Timeout*.

The server announces to all connected clients that it is waiting for the submission of a valid solution with maximum number of *Moves* moves from player *Player*. It will accept this solution within *Timeout* moves. With this command, the server also announces it is in the **After** phase. While all players receive this command, only the player mentioned is requested to submit a solution using the `Solution` command.

`Winner` *Winner*.

The server announces to all connected clients the winner of the current game and the solution that won, or that there has been no winner. In any case, the `Winner` command marks the end of the **After** phase and the start of a new **Before** phase. The server will go on to immediately send a new *Board* command to all connected clients.

Context-free grammar

In this part, the syntax of the command arguments is specified.

```
Board    ::=Pos Walls Lambdas
Walls    ::=[ ] | [ (Wall ,)* Wall ]
Wall     ::=( Pos , Dir )
Lambdas  ::=[ ] | [ (Lambda ,)* Lambda ]
Lambda   ::=( Color , Pos )
Moves    ::=Nat
PlayerId ::=Nat
Player   ::=String
Players  ::=[ ] | [ (Player ,)* Player ]
Prob     ::=Color Pos
Solution ::=[ ] | [ (Step ,)* Step ]
Step     ::=( Color , Dir )
Timeout  ::=Nat
Winner   ::=None | Player Solution
Color    ::=Red | Green | Blue | Yellow
Dir      ::=N | E | S | W
Pos      ::=( Nat , Nat )
```

A *Board* is specified by its size (in the form of a coordinate pair *Pos*), a list of *Walls* and a list of *Lambdas*. Lists are in Haskell syntax, surrounded by square brackets, the items separated by commas. Pairs are also in Haskell syntax, surrounded by round brackets and the two components separated by a comma.

A *Wall* is a pair of a *Pos* and a `Dir`. It indicates that moving from position *Pos* in direction `Dir` is blocked. Note that this allows for boards to have walls that only block movement in one direction. You may want to prepare your client for that possibility, and have it show unidirectional walls differently from normal, bidirectional walls. The walls surrounding the whole board are implicitly present and may be contained in the list of walls, but do not have to be.

A *Lambda* is a pair of a *Color* and a *Pos*, indicating that the Lambda of that color is of that position. In the list of Lambdas, the colors that occur should all be different, but not all problems have to use all available colors.

A specification of *Moves* is just a natural number `Nat`.

A *PlayerId* is a natural number `Nat`. Player names *Player* are given as a *String*. Finally, *Players* are a list of *Player*.

A *Prob* is given by a *Color* followed by a *Pos* as well, indicating that the Lambda of the color *Color* has to be moved to position *Pos*. Obviously, the server should only pose problems for a Lambda color that exists on the current board.

A *Solution* is given by a list of `Steps`. A `Step` is a pair of a *Color* and a `Dir`. This means that the Lambda of the appropriate color has to be moved in direction `Dir`. A solution is valid if after performing the steps in order, starting from the initial positions of the Lambdas on the board, the problem is solved, i.e., the Lambda of the specified color is on the specified square. The length of a solution is given by the length of the list of steps.

A *Timeout* is a natural number `Nat`, specifying how many seconds are left in the game.

For *Winner*, there are two alternatives. The terminal `None` indicates that there is no winner. Otherwise, the winning *Player* and the winning *Solution* are given.

All tokens in the above grammar may be separated by spaces (not spread across many lines though, as each message takes exactly one line).

Lexical syntax

Natural numbers `Nat` are a non-empty sequence of digits. Strings *String* are sequences of characters surrounded by double quotes. In its simplest form, strings may only contain 7-bit ASCII characters and must not contain either backslashes or double quotes. Haskell-like backslash-escaped characters may be added.

4.2.4 Additional requirements

In this part, several additional requirements of both the client and the server component are given.

The client

The server and port the client tries to connect to should be configurable (via command line, preferably).

The client should keep track of the players currently in the game. This information is only available by following the announcements of the server in the `AcceptedPlayer`, `PlayerGone` and `Problem` commands.

During the game, the client should keep track of the bids that have been placed by the players, by keeping track of the `ReceivedBid` announcements. At the very least, it should be obvious to the player at any point of the game who has placed the best bid so far.

The client should provide a graphical representation of the board. The client should not rely on the board always having the same, or a specific, size. The client should provide a way to edit and submit a solution while playing the game, and it should provide a way to display the winning solution after it has been announced using `Winner`.

The client should make gaming convenient. Make it easy to place a bid even before entering a solution. Placing a bid fast is important to win the game, so it should not be complicated to do so.

Make it easy to pick a name for the player and use the same name later on in subsequent games.

The server

The server should continuously stick to the **Before**, **In**, **After** loop. It should always accept new connections, and should handle each connection in its own thread. The port on which the server listens for incoming connections should be configurable (via command line, preferably).

The server has a certain amount of flexbility in the generation of timeouts: there should be a point where a server switches from **Before** to **In** phase, even if not all players currently connected have confirmed. Non-responsive players shouldn't be able to spoil the fun for all the others. One option is to have the first `Confirm` start a timeout of, say, one minute during which other players can confirm. If all connected players confirm earlier, the game can start earlier.

Also, the server has to generate a timeout for a game initially. If a puzzle turns out too hard to solve, or if players all lose interest and quit the game or become non-responsive, the server should not stay in the game forever (during this time, other newly connected players will have to wait, after all). The demo server uses 300 seconds for this purpose, but you can choose another value.

After the first bid – at least in the standard version – the timeout should be reset to 60, and even if it previously has been less than 60. From then on, the timeout is unaffected by further bids.

The server should provide some variation in the boards and the problems. The demo server currently uses only one board, but many problems on that board. You can start with that board, which is one of the board game boards – it works well. But try to also allow a few different boards. Think about the characteristics that make a board well-suited for the game and explain/motivate your choices.

When generating problems, pay attention to the fact that the problem should be solvable. Not all positions on the board may be reachable. In particular, positions too far away from walls or even corners may be very difficult to reach.

The server should not pose problems that are solvable in less than four moves with a single robot. The demo server currently does not have this feature.

4.2.5 Potential extensions

Once you have a client and a server that adhere to the minimal requirements above, you can start implementing extensions. Here are a few ideas, but you should not feel limited by them.

Exercise 4.2.1. Implement an automatic or semi-automatic client. Either try to write a fully automatic client that solves problems without user interaction and tries to be competitive with interactive clients (or ideally is much faster). Alternatively, you can at least try to solve easy problems automatically, or support the user with hints.

Exercise 4.2.2. Different numbers of Lambdas. Instead of a maximum of four, allow five, or many more. Vary the number of make it configurable.

Exercise 4.2.3. Allow problems where *any* Lambda may be moved to the target square.

Exercise 4.2.4. Implement random board generation. Try to generate good boards and problems that are neither too easy nor too difficult.

Exercise 4.2.5. More board features (really use unidirectional walls, add fields that change the direction of (some) Lambdas, that shift Lambdas one position to the side, that teleport robots to some other place etc.).

Exercise 4.2.6. Change the physics (for instance, if a Lambda hits another Lambda, the impulse from the first carries over to the second).

Exercise 4.2.7. Change the scoring system (for instance, in order to encourage solutions that use many Lambdas, parallel moves of different Lambdas could just count as a single move).

Exercise 4.2.8. Implement more variants of the game, or even completely different, but somewhat similar games. Make the game variant choosable by the clients, or configurable when starting the server.

Exercise 4.2.9. Make the server more verbose and responsive. Instead of just ignoring commands it does not understand or that come in the wrong phase of the game, it could respond with appropriate informative messages about what is going on or what it is expecting.

Exercise 4.2.10. Make it possible for players logged into the game to communicate (chat) with each other.

Exercise 4.2.11. Allow multiple games on the same server being carried out in parallel (i.e., have multiple game rooms).

Exercise 4.2.12. Collect statistics about won or lost games in the server and make those statistics available by querying the server. Player data should be stored in a text file or a database.

4.3 LambdaRogue (***)

The goal of this assignment is to reimplement a simple "roguelike" game in Haskell. This specification lists basic requirements as well as ideas for extending the game beyond the minimal requirements. Also included are a few tips and tricks for the implementation.

4.3.1 Demo game

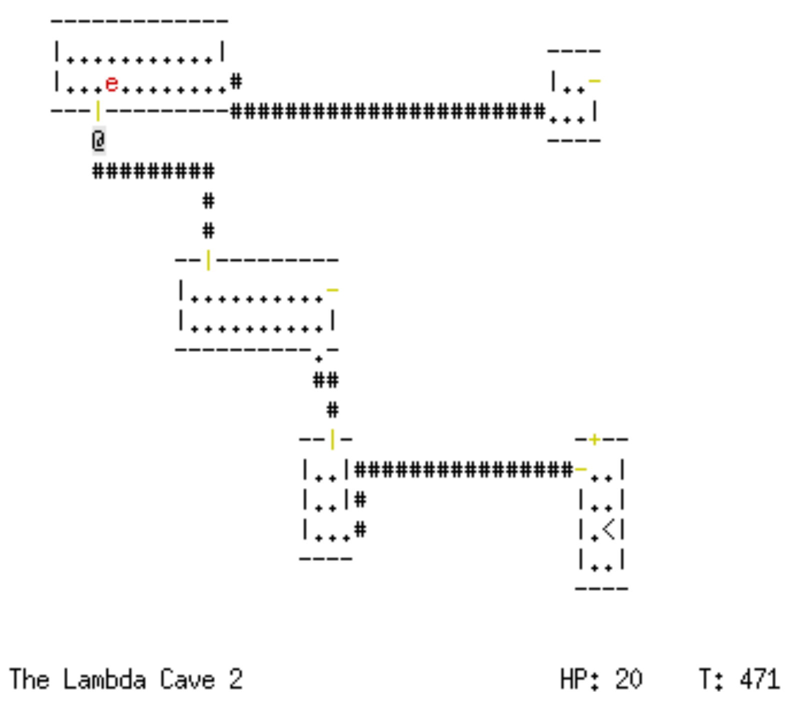
For demonstration purposes I have written a demo game that more or less demonstrates the requirements from this task description. It is, however, *not* a reference implementation. I might extend or change the demo throughout the course and implement a few additional features myself, so it is probably not a good idea to rely on the demo too much.

Nevertheless, the demo game may serve to understand the task description better and to experiment a bit with a small and simple game.

Of course, it is also recommended to try other roguelike games for ideas.

The demo game is available by calling `LambdaHack` from the command line on the lab machines.

Here are a few keys you can use in the demo game:



```
-------------
|...........|                       ----
|...e........#                      |..-
---|---------#######################...|
   @                                ----
   #########
           #
           #
         --|---------
         |.........-
         |.........|
         ----------.-
                   ##
                    #
                  --|-                -+--
                  |..|################-..|
                  |..|#               |..|
                  |...#               |.<|
                  ----                |..|
                                      ----


The Lambda Cave 2                    HP: 20    T: 471
```

Figure 4.1: Demo game screen shot

key	command
`k`	up
`j`	down
`h`	left
`l`	right
`y`	up-left
`u`	up-right
`b`	down-left
`n`	down-right
`<`	level up
`>`	level down
`S`	save and quit the game
`Q`	quit without saving
`o`	open a door
`c`	close a door
`s`	search for secret doors
`.`	wait
`,`	pick up an object
`i`	show what you are carrying
`:`	look around
`v`	display the version of the game
`V`	toggle field of vision display
`O`	toggle "omniscience"
`M`	display level meta-data
`R`	toggle smell display
`T`	toggle level generation sequence

Pressing a capital letter corresponding to a direction key will have the character run in that direction until something interesting occurs.

4.3.2 Dungeon

Dungeon layout

The player should be able to explore a dungeon. The dungeon consists of multiple levels (at least 10), and each level consists of at least 80 by 21 tiles.[1]

At least the following tiles should be implemented:

tile type	symbol in demo game
floor	`.`
wall (horizontal and vertical)	`-` and `|`
corridor	`#`
stairs (up and down)	`<` and `>`
rock	invisible

The player is able to move around floors, corridors and stairs, but not through walls or rock.

The game world should be persistent, i.e., every time a player visits a level during one game, the level should look the same.[2]

Level generation

Each level is generated by an algorithm inspired by the original Rogue, as follows:

- The available area is divided into a 3 by 3 grid where each of the 9 grid cells has approximately the same size.
- In each of the 9 grid cells one room is placed at a random location. The minimum size of a room is 2 by 2 floor tiles. A room is surrounded by walls, and the walls still have to fit into the assigned grid cells.
- Rooms that are on horizontally or vertically adjacent grid cells may be connected by a corridor. Corridors consist of 3 segments of straight lines (either "horizontal, vertical, horizontal" or "vertical, horizontal, vertical"). They end in openings in

[1]This minimal size goes back to a standard terminal size of 80 columns and 25 lines. A few of the lines are used to display status information, the rest is available to display the level.

[2]This may sound obvious – I'm only saying that because there *are* roguelike games that don't allow you to go back up, or that generate a new random level every time you change levels.

the walls of the room they connect. It is possible that one or two of the 3 segments have length 0, such that the resulting corridor is L-shaped or even a single straight line.

- Corridors are generated randomly in such a way that at least every room on the grid is connected, and a few more might be. It is not sufficient to always connect all adjacent rooms.
- Stairs up and down are placed. Stairs are always located in two different randomly chosen rooms.

The algorithm should be written in such a way that it can easily be applied to other map and grid sizes.

Figure 4.2 visualizes a sample level generation.

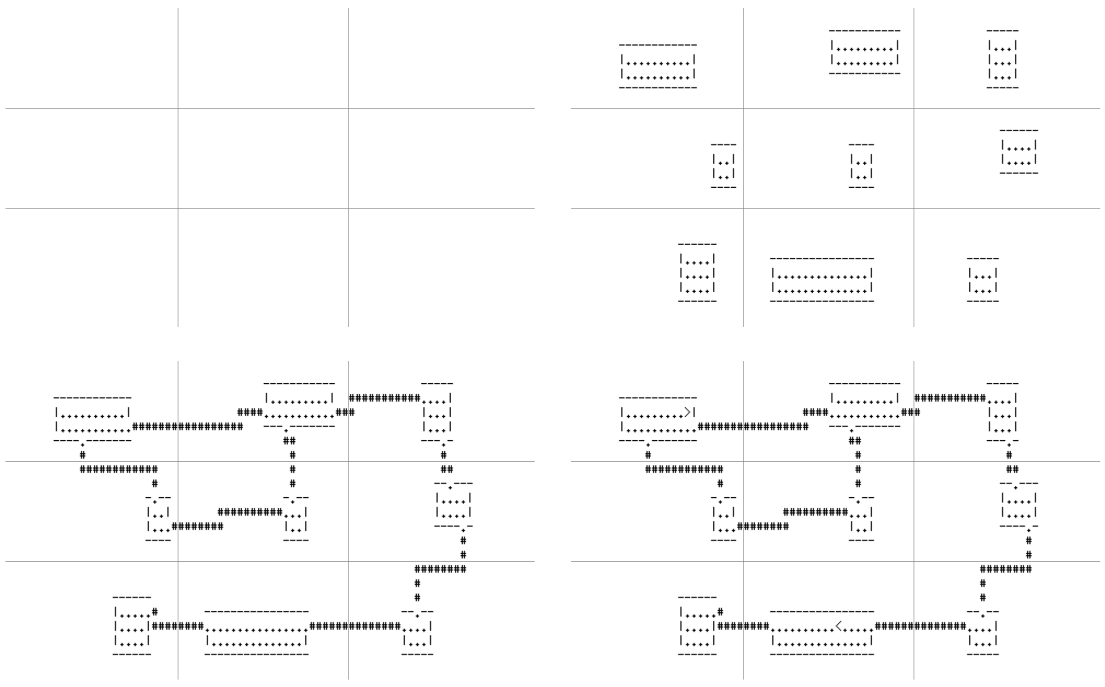


Figure 4.2: Level generation in four steps: first (upper left) we have an empty grid; we then (upper right) generate a room in each cell; then (lower left) corridors are added; finally (lower right) stairs are placed

4.3.3 The player

Movement

The player (`@` in the demo game) should start the game on the staircase up of the first dungeon level. The player should be able to move around horizontally, vertically, and diagonally in the dungeon, and should be able to change levels at staircases.

It should be possible to direct the player using keyboard or mouse. Just for reference, the demo game uses the Rogue keybindings for player movement, which in turn are inspired by the editor Vi. The list is given in Section 4.3.1.

Status

Add a status line that displays some current information about the player's situation in the game world, such as the current level, or the player's health.

Vision and memory

Only parts of the game map that the player has already explored should be shown. Only parts othe the map that the player can currently see should be shown as they currently are, other parts should be shown as the player remembers. This requires to implement an algorithm that determines what the player can see. Intuitively, the player can see a tile in the map if the player has an unobstructed line of sight to that tile.

We first specify fields that are reachable from the player. As input to the algorithm, we require information about fields that block light. As output, we get information on the reachability of all fields. We assume that the player is located at position `(0,0)`, and we only consider fields `(line,row)` where `line` >= 0 and $0 \leqslant$ `row` $\leqslant$ `line`. This is just about one eigth of the whole player surroundings, but the other parts can be computed in the same fashion by mirroring or rotating the given algorithm accordingly.

```
fov (blocks,maxline) =
  shadow        :=$\emptyset$
  reachable (0,0):=True
  for l ‘elem‘ [1..maxline] do
    for r ‘elem‘ [0..l] do
      reachable (l,r):=($\exists\alpha . \alpha$ ‘elem‘ interval (l,r) && $\alpha \notin$ shadow)
      if blocks (l,r) then
        shadow:=shadow $\cup$ interval (l,r)
      end if
    end for
  end for
  return reachable
```

```
interval (l,r) = return [angle (l + 0.5, r − 0.5), angle (l − 0.5, r + 0.5)]
angle (l,r)    = return atan (r / l)
```

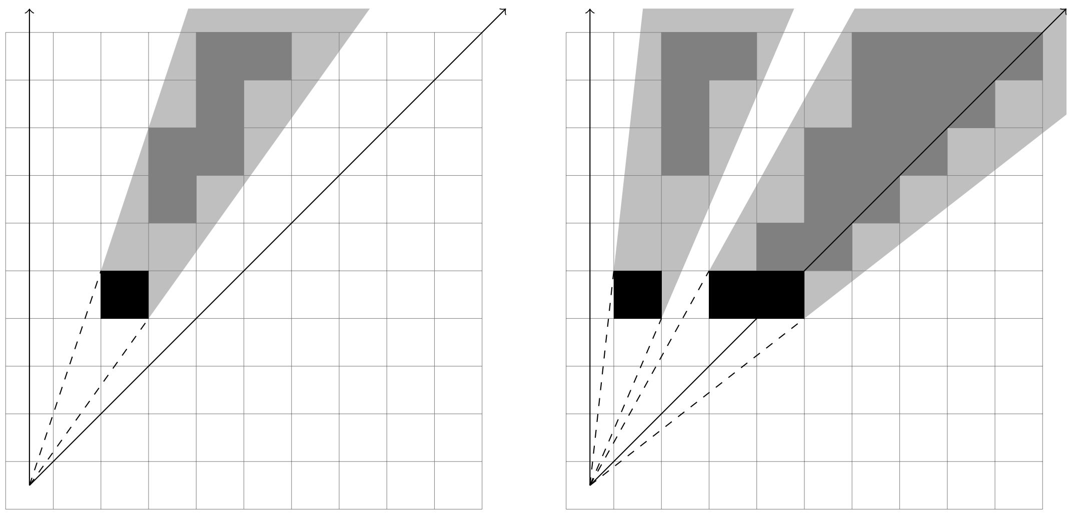


Figure 4.3: Visualization of the FOV algorithm

Look at Figure 4.3 for visual help. Lines are depicted upwards, rows to the right. The algorithm traverses the fields line by line, row by row. At every moment, we keep in `shadow` the intervals which are in shadow, measured by their angle.

A square is reachable when any point in it is not in shadow – the algorithm is permissive in this respect. We could also require that a certain fraction of the field is reachable, or a specific point. Our choice has certain consequences. For instance, a single blocking field throws a shadow, but the fields immediately behind the blocking field are still visible, as can be seen from the left example image.

We can compute the interval of angles corresponding to one square field by computing the angle of the line passing the upper left corner and the angle of the line passing the lower right corner. This is what `interval` and `angle` do.

If a field is blocking, the interval for the square is added to the `shadow` set.

Optimization Note that you will probably have to optimize the algorithm. For instance, the algorithm expects an argument `maxline`. Generally, it is possible to stop the computation once the whole range we consider is in shadow, even before we reach a given maximum line. Also, it pays off to choose a suitable representation for `shadow` that joins intervals and only remembers the points where light changes to shadow and shadow changes to light.

Light

Once you can compute the reachable fields using fov, it is possible to compute what the player can actually see.

Fields adjacent to the player (also diagonally) can always be seen (except for walls, see below). Fields that have light and are reachable can also be seen. We treat floor of rooms as having light, whereas corridors and rock are dark.

Light and walls

Walls reflect light. They can be seen only if an adjacent floor field can also be seen. In particular, walls cannot be seen when passing a corridor on the outside of a room, but can be seen from the inside of a room.

4.3.4 Saving games

It should be possible to save the game to a file, and restore from there. Everything about the game should be stored, so that restoring continues in exactly the same situation.

As most roguelikes, you should remove the save game after successfully restoring from it.

4.3.5 Monsters

The player is not alone in the dungeon. Monsters should roam the game world, too.

Monsters inhabit a specific location on the game map, and can be seen if the field they are on can be seen by the player. Monsters are not remembered, i.e., they are removed from the display once the player can no longer see them. [3]

Monster generation

Monsters are generated in random time intervals. Of course, monsters should only be generated on empty squares (no other monsters, no player, no walls, no rock). It

[3] This would be more confusing than helpful, because monsters can move. An exception might be a graphical UI where you have a good way to distinguish remembered monsters from seen monsters.

might also be a good idea never to generate monsters in sight of the player, because the sudden appearance might be confusing.

Monster movement

Monsters should move. Every monster gets a turn per move of the player. Monster moves are restricted in the same way as player moves, i.e., they cannot move into obstacles like walls or rock.

You should implement some algorithms that the monsters can use in order to find the player. Not all monsters have to use the same algorithm, but the following features should be available for generated monsters to select from.

Random The simplest way to have a monster move is at random.

Sight If a monster can see the player (as an approximation, you can say that this is the case when the player can see the monster), the monster should move toward the player.

Smell The player leaves a trail when moving toward the dungeon. For a certain timespan (100–200 moves), it should be possible for certain monsters to detect that a player has been at a certain field. Once a monster is following a trail, it should move to the neighboring field where the player has most recently visited.

Noise The player makes noise. If the distance between the player and the monster is small enough, the monster can hear the player and moves into the approximate direction of the player.

More Flesh out the above algorithms. For instance, try to have monsters surround small obstacles automatically, switch between senses depending on whether a monster can see, smell or hear the player, or make the random movement less random by rembmering the direction the monster last moved in, and not moving into the opposite direction during the next move.

Combat

When the player moves into a monster, or a monster moves into the player, combat occurs. You should invent at least a minimal system that keeps track of each monster's and the player's health. Whenever combat occurs, the attacked party loses some health.

If the player dies, the game ends.

If a monster dies, it is removed from the map.

Monsters should not normally fight each other (even though in the demo game, they do).

Messages

Combat requires information to be shown to the user. The player should be informed if he is attacked or attacking a monster, and what the outcome of the battle is. Make sure that only information is displayed that is relevant to the player.[4]

4.3.6 Extensions

Here, a few possibilities to extend the game beyond the minimal requirements are listed. The more, the better. These are just ideas. Feel free to design your game differently or to implement an extension that is not in this list.

Items

Have random items in the dungeon levels (or possibly, even a few non-random items). Treasure, magic items, weapons or tools are all options. Implement the ability to pickup or drop items, and keep track of the inventory of items carried by the player.

Allowing to use items in various ways adds a lot of depth to the game.

Can monsters make use of items as well?

Score

Award the player points for things achieved in the game (survival, beating monsters, reaching new levels, finding treasure, ...). Implement a high-score list where the best scores of each player are stored. Of course, you should ask for the name of the player at one point.

Typically high-score lists of roguelike games list the cause of death.

Doors and other dungeon features

Add doors that can be open or closed. Doors might also be secret so that you have to search for them before being able to use them.

[4]If you share code between monsters and the player, there is a danger that messages for the monster a displayed as a message to the player.

Other terrain types are possible in dungeons: you can have (movable) boulders (that might block sight), water, lave and more ...

Graphical user interface

There are lots of options in this area. You can go for a tile-based graphical interface that draws little pictures rather than ASCII graphics. You could even animate the tiles.

You can also go for a first-person 3D interface.

Or you can just spice up the text-based user interface, by adding status information or displaying messages in message windows.

Proper combat system

Add chance as a factor to combat. Have different monsters and the player do different amounts of damage, and have a different chance to inflict damage on the opponent. If you have weapons and armor in the game, make it possible to equip a better weapon in order to inflict more damage, or to wear a better armor in order to gain more protection from attacks.

What about ranged combat? You might allow bows and arrows in your game, that can be shot over a distance. Or you could have dragons that can breathe fire from far away ...

Magic effects

Whether magic or not, a lot of effects can be implemented: fireballs or other combat help, but also the possibility to gain information: find out the complete map, or discover the whereabouts of items and valuables. Have items that trigger those effects, or allow the player to cast a limited number of spells in a certain amount of time (i.e., turns).

Larger levels

There is no need to limit the level size to the size of the screen. You can make levels that are very large, and use scrolling (even in text mode) to show the portion of the level where the player currently is.

Improved level generation

Experiment with different level generation algorithms that add dungeon levels with different characteristics: mazes, caverns, cities, strangely shaped rooms – even wilderness or underwater levels are possible . . .

Food

Have the player need food to survive. Food can be found in the dungeon, or possibly dead monsters can be eaten. If the player does not get enough food, the player loses health or simply starves to death.

Speed

Different monsters have different speed, and can thus make moves more often or less often than the player. The player can speed up and down through items or other means.

Story

A story adds flavour to the game. Have a background story and a goal to achieve. Add all sorts of tasks and subtasks the player has to solve. Quests may require certain non-random items or rooms to be placed in the dungeon.

4.3.7 Advice

Keep IO to a minimum

Only put code into the IO monad that really has to be. Separate out the user interface from the game logic. Many parts of the program need random numbers. Write your own monad for random numbers if you want an abstraction, but don't use the IO-variants of the functions.

Not using IO much also makes your program easier to test.

Find the "right" level of abstraction

Don't generalize too much in the beginning, because it will slow you down. But also try to think about possible generalizations and how difficult they would be with your code, so that you can keep the amount of rewrites needed low. Document design decisions so that you and your teammates will remember them yourself later on.

Use the right data structures

Don't use functional (i.e., immutable) arrays! This is a performance killer. There are at least two suitable representations for level data: a finite map from locations to tiles (`Data.Map`) or *mutable* arrays (`Data.Array.ST` or `Data.Array.IO`). The latter are potentially more efficient, but also have the disadvantage of at least forcing monadic structure of even IO onto large parts of your program.

Intersperse development with refactoring phases

Whenever you've successfully implemented a feature, sit back for a moment and think about how to clean up and restructure your code – *before* you start on the next feature. Trying to restructure or clean up everything at the end may be too late.

Add debugging features

Add a lot of debugging features to your game. Different ways to display things and the possibility of showing meta-information can help both you and me to debug your game. Definitely make sure that the field-of-vision algorithm, the smell, and the complete level plus locations of monsters can be visualized.

4.4 Asteroids (∗∗∗)

In this assignment you'll be developing a small game in Haskell. This assignment will be a bit more "realistic" than the previous assignments: the program you'll write has to respond in real-time to user input, will consist of multiple modules, and you have more freedom in how and what to implement.

4.4.1 Introduction

In this assignment you'll make use of the Gloss graphics library for Haskell, which provides a very high-level interface for drawing graphics on screen and handling user input.

The game you'll be implementing is a variant of the classic 1979 arcade game *Asteroids*.

Getting started

Either download and extract the starting framework from the course website or clone the assignments repository from Github.

The starting framework contains two folders:

framework This folder contains the modules which you need to modify in order to implement the minimal requirements, as well as the cabal file necessary to build the game.

executables This folder contains executables for Windows, OS X and Linux of a version of the game that already implements all of the minimal requirements.

The Gloss library

The home page of Gloss can be found at `http://gloss.ouroborus.net/`. It contains some instructions on how to install the Gloss library and solve common problems. On the machines we tested the library with (including those at the university's computer lab) we only needed to run the command:

```
cabal install gloss
```

More interesting is the library's documentation, which can be browsed at `https://hackage.haskell.org/package/gloss-1.8.1.2`. In particular you'll likely be interested in the modules:

Graphics.Gloss (functions that create the main window and handle events);

Graphics.Gloss.Data.Picture (combinators for drawing pictures);

Graphics.Gloss.Data.Color (helper functions for working with colors);

Graphics.Gloss.Geometry.Angle (helper functions for converting between degrees and radians).

Finally, you may want to have a look at `https://hackage.haskell.org/package/gloss-examples`, a package which contains numerous example program written using the Gloss library.

Compiling and running the starting framework

The compile the starting framework, go into the `framework` folder (not the `src` folder!) and type:

```
        cabal install
```

To run the program, type:[5]

```
lambda-wars
```

At this point the program will crash, because not all the required functions have been implemented yet.

4.4.2 Overview

The game is a variant of the game *Asteroids*. The player controls a space ship that can move through a region of space, shoot at enemies to gain points, and pickup bonus objects that increase the score multiplier. This is all accompanied by some exciting visual effects.

Modules

The starting framework follows the model–view–controller pattern. The game is divided into the following modules:

Model This modules contains the data type definitions that are used to represent the game state.

View This module uses the game state to render a picture.

Controller.Event This modules handles keyboard events (by queuing them in the game state). You will not have to change much, if anything, in this module.

Controller.Time This modules specifies what needs to be done on each frame update. It handles the queued input events and updates all state that needs to evolve with time.

4.4.3 Requirements

The requirements are separated into *minimal requirements*, those that you have to implement in order to receive a passing grade—assuming your coding style is not too awkward, and *optional requirements* that you can implement in order to receive a higher grade. Grades do not scale linearly with the amount of features implemented: in order to improve your grade further, you will have to do increasingly more work.

Also look at the supplied executables of the game. They may make the requirements clearer than the textual requirements given here.

[5]Or "`lambda-wars` *`width height`*" to run the game in full screen mode.

Minimal requirements

Player movement The player's space ship should be able to rotate to the left and right, and be able to thrust forward.

Enemy spawning and movement Enemies should spawn randomly in space and move towards the player's ship. If an enemy touches the player's ship, the ship should blow up and the score multiplier be reset to one.

Shooting The player should be able to shoot. If a bullet hits an enemy or bonus object that enemy or object should be destroyed.

Score keeping The score multiplier should increase by one for each bonus object the player picks up, and the score itself should be increased by the score multiplier for each enemy the player shoots.

Particle effects If the player's ship is destroyed it should explode using a nice visual effect. If the player thrusts, an exhaust trail should be left behind.

Background A star field should be drawn in the background. Stars should have depth, which is made visible by *parallax scrolling*.

Optional requirements

Here are some suggestions for additional features that you can add to the game. You can also come up with your own ideas.

More enemy types Add multiple types of enemies with various kinds of appearance and behavior. Spawn them in an interesting pattern.

High scores Add a high score table to the game that is saved and read from disk. This obviously also requires limiting the number of lives a player has.

Multi-player Add a multi-player mode to the game. You will probably want to have a look at the Network, Network.Socket or one of the other networking libraries on Hackage.

Menu Add a menu to the game that allows you to view the high scores and/or select the number of players, difficulty setting or level. It is probably easiest to model this as some kind of finite automaton, so you can keep a nice separation between model, view and controller.

4.4.4 Hints

Record syntax

The game state is represented as a record. To keep your code elegant you should know how to work with records effectively. Haskell has some convenient syntax for working with records, especially if you have enabled the `NamedFieldPuns` and `RecordWildCards` language extensions (as has been done in the starting framework).

Pattern matching on a record To pattern match on a record and bring two of its field into scope, write:

```
foo (World {field1, field2}) = bar field1 field2
```

To pattern match on a record and bring all of its field into scope, write:

```
foo (World {..}) = bar field1 field2
```

To pattern match on a record and also give the whole record a name at the same time, write:

```
foo world@(World {field1, field2}) = bar' world
```

"Updating" a record To update some fields in a record, write:

```
foo world@(World {field1, field2, field3})
   = world {field1 = 42, field2 = bar field1}
```

Note that the argument passed to `bar` is the original value of `field1` in the record `world`, and not the value `42` in the newly constructed world object!

Accessing a field that is not in scope If you need to access a field that is not in scope, write:

```
foo world@(World {field1}) = bar field1 (field2 world)
```

Advanced approaches

Instead of using records to keep track of your game state, there are also some other approaches that may work. These are more advanced techniques and not necessary to be able to get the game working. We mention them here, but we'll leave it up to you to figure out how these approaches work for yourself.

Monads Instead of passing the game state explicitly, you can hide it inside an appropriate (state) monad and abstract the state updates with a suitable monadic interface. This has the advantage that some parts of the state (such as the random number generator's seed) can be updated automatically by the monad, while you only have to worry about the important parts. The potential disadvantage is that this will give your program a much more imperative feel.

Lenses Record syntax works reasonably fine when you have flat records. Once you start working with nested records, it can quickly become a pain, though. One of the solutions proposed for this problem are lenses. There are several implementations of lenses: `https://hackage.haskell.org/package/lens` is one of the more popular ones, but has a very steep learning curve; `https://hackage.haskell.org/package/data-lens` and `https://hackage.haskell.org/package/fclabels`[6] are less powerful, but easier to use.

Some tutorials on working with lenses can be found at:

- `http://www.haskellforall.com/2012/01/haskell-for-mainstream-programmers_28.html`
- `https://www.fpcomplete.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial`
- `http://fvisser.nl/post/2013/okt/1/fclabels-2.0.html`

But again, this is all advanced stuff and not necessary for you to know or use in order to be able to finish the assignment.

Cabal

If you want to add extra modules to the framework, you will also have to list them in the cabal script (`framework/lambda-wars.cabal`). For more information on cabal, see `http://www.haskell.org/cabal/users-guide/developing-packages.html`.

[6]Written by former students from Utrecht University, who now work at a start-up that develops web applications using Haskell.

5 Larger tasks: web programming

5.1 Blog server, Snap based (∗∗)

The goal of this assignment is to extend an existing web server for blogs in order to obtain experience with a 'real world' application. The web server is constructed on top of the Snap framework but also uses many other packages and GHC specific extensions. Building a larger application usually involves the use of many of such (different) language ingredients, understanding these to a reasonable degree and letting these work together usually is the challenge of such larger applications where coding is not done from scratch but done by composition of existing libraries.

The web server named `blog-server` can be found at:

`https://github.com/atzedijkstra/blog-server`

The `README` associated with the projects contains references to the various used packages.

5.1.1 Getting started

Download the web server code, build the web server using `cabal`, preferably using a sandbox (`http://coldwa.st/e/blog/2013-08-20-Cabal-sandbox.html`) as many specific libraries are used with whom existing versions may clash.

```
> git clone https://github . com / atzedijkstra / blog − server
> cd blog − server
> cabal sandbox init
> cabal install  -- force-reinstalls
```

The `--force-reinstalls` may be necessary when newer versions of libraries are installed in the sandbox on which other libraries may depend. After locally installed, unpack the example state, run the server and startup a browser at `http://localhost:8000`.

```
> tar xfz state . tgz
> .cabal − sandbox / bin / blog − server − p 8000
```

See the `README` for what the provided state offers. Using the provided state is not necessary, then the server starts building state from scratch.

5.1.2 User info (authentication, html rendering)

The server currently does not show when a user was logged in for the last time.

Exercise 5.1.1. Explore the `User` datatype which holds a `AuthUser` from the snap framework authorisation infrastructure to find out where info about 'last time logged in' is maintained. The relevant fields may or may not be filled correctly, adapt the server to show this info in the editing of the user settings (menu `My stuff`) and on every page where a user is logged in.

Exercise 5.1.2. Explore the `User` datatype which holds a `AuthUser` from the snap framework authorisation infrastructure to find out where info about 'last time logged in' is maintained or might be added.

5.1.3 Full state rendering (programmatic html rendering)

Via menu `Dump adm` a text based dump of the internal state can be generated, but much of the tabular nature of this info is not shown.

Exercise 5.1.3. For html rendering the `blaze` library is used. The current rendering of (e.g. `Users`) state is done by embedding the pretty printing of such state in a html `pre` tag. Adapt this by programmatically rendering all info inside tables.

5.1.4 Domain model changes (data storage, persistency)

The persistency library `acid-state` uses the library `safecopy` for storing data. The `safecopy` library provides mechanisms for allowing a domain model (read: datatypes) to change whilst old state still can be used and be adapted on-the-fly. In order to make this work, `safecopy` knows about domain model versions and requires explicit functions to be implemented for transforming datat between versions.

Exercise 5.1.4. Blog entries are shown in "last created shown first" order, based on the increasing sequence number assigned as an identification to each blog entry. Adapt the `Blog` datatype to include a modification date, and adapt the rendering of blogs to show blog entries in "last modified shown first". Explore the `safecopy` library on how to deal with mentioned domain model changes.

Exercise 5.1.5 (optional, difficult)**.** Modify the rendering of logs to be able to choose between different orderings, e.g. ascending/descending, and/or ordering based on modification time/creation time/user name/etc.. Make this choice part of a (persistent) user preference. In order to make this work the whole application and its user interface generation needs to be explored and modified.

5.1.5 Automatic checkpoints (concurrency)

The `acid-state` library is based on a combination of

- (checkpoint) storing a full in-memory state onto disk (i.e. various files), and
- (event) logging modifications to this full state.

In case of failure the last full state is read and the events (function calls) from the log are re-run. This is mechanism many transaction based systems use, but this mechanism requires a balance between the time consuming store of full state (which may be very large) and the re-run of the log.

Exercise 5.1.6 (difficult)**.** Although creating checkpoints (function `createCheckpoint`) can be done manually (menu `Sync adm`), this can better be done automatically, for example during server idle time. Explore the `snap` framework for its use of threads (see `GHC.Conc`) and programmatic ways of finding out whether server threads are idle. You may have to use `https://wiki.haskell.org/ThreadScope`. If possible, add a thread which its sole purpose is to monitor the idle state of the server, and if idle enough performs a checkpoint (but not too often either, and perhaps not if not much has to be stored).

5.1.6 Database access (sql wrapping)

The persistent storage is not done via relational databases, too keep the server relatively simple. The snap framework offers various plugins for accessing databases, e.g. `snaplet-hdbc`, or `snaplet-mysql-simple`.

Exercise 5.1.7 (optional, difficult)**.** Replace the persistency layer based on `acid-state` by one based on a database layer of your choice.

5.1.7 Blog upload (parsing)

Usually some external representation of otherwise internal data can be uploaded to a website. Such a representation can be CSV (Comma Separated Values) or a variant thereof; or it might be another format of your choice.

Exercise 5.1.8. Design and/or choose an external representation for multiple blogs. Write a parser for it, and use this parser to add upload functionality for blog entries.

5.1.8 More functionality

Exercise 5.1.9 (optional)**.** Extend the server to you liking, e.g. by adding blog categories, tags, etc.. Beef up the rendering by properly using style sheets (if you happen to be an expert in that, or want to be...).

5.2 Debt tracking server, Yesod based (***)

In this assignment you will develop a small web application for keeping track of who owes you money and who you owe money to, called IOU (pronounced "I owe you").

You should read this text completely and carefully before beginning to implement the application.

5.2.1 Introduction

In this assignment you'll make use of the Yesod web framework for Haskell, which will handle a lot of the details of writing a web application for you.

Getting started with Yesod

An online version of the book *Developing Web Applications with Haskell and Yesod* can be found at `http://www.yesodweb.com/book`. You will need to at least skim the chapters in the "Basics" part of the book and have enough familiarity with Yesod so that you understand how the blog in the "Examples" part works. This application has a fairly similar architecture to the one you will be developing. You don't need to know any of this for the exam, though.

In order to complete this assignment you will only need a minimal amount of knowledge about HTML (if you know what `<a>`, `<p>`, `<h1>`, `<ul>` and `<li>` mean, then you should be fine, otherwise it will likely become clear by reading the book, as well) and probably no knowledge of CSS and JavaScript at all.

Some basic knowledge about relational databases is helpful. While you don't need to write any queries in SQL, you do need to know what a relational database schema looks like and what a join is. If you didn't take the course on Databases, then you might want to team up with someone who did, or do some extra background reading.

Installing Yesod

To install the Yesod web framework on your machine, issue the following Cabal commands from the command line:

```
cabal update
cabal install alex
cabal install yesod-platform
cabal install yesod-bin
```

Running the starting framework

After you have extracted the starting framework, you can open a command line, go to that folder, and start the framework by running the commands:

```
cabal install
yesod devel
```

The final command will start the "development server" that will monitor any of the files in the starting framework and automatically recompile them for you when they are modified. You can visit the web application by opening the URL `http://localhost:3000/` in a web browser.

5.2.2 Overview

The motivation for developing this application is as follows: when eating with a group in a restaurant or ordering some pizzas online, it's often more convenient if one person pays the bill and the other people promise to repay that person at a later time (either by giving him or her some cash, or by paying for the pizza bill the next week).

When eating with a large group, or when ordering pizzas online often enough, it can-become quite a hassle to keep track of all those debts that need to be repayed, so clearly this administration would benefit from some automation.

Entities

The following entities are of importance to the application:

Users A User is anyone who has ever logged in on the system. Users can add new Receipts and Payments into the system and be added to Receipts as debtors.

Receipts The User paying the bill to the restaurant owner or pizza delivery guy will receive a receipt. He or she can enter this Receipt into the system, specifying the amount they have payed and which Users will have to repay part of the bill to them.

Payments Occasionally a debt will be settled by the exchange of money between two Users. These transfers will also have to be administered in the system. A Payment simply denotes the exchange of a certain amount of money from one User to another User.

5.2.3 Requirements

The requirements are separated into *minimal requirements*, those that you have to implement in order to receive a passing grade—assuming your coding style is not too awkward, and *optional requirements* that you can implement in order to receive a higher grade. Grades do not scale linearly with the amount of features implemented: in order to improve your grade further, you will have to do increasingly more work.

Minimal requirements

Your application should respond to the following URLs:

`/user` Should list all users that have logged in to the system, linking to their `/user/`*`userId`* page.

`/user/`*`userId`* Should for a given user display:

- The receipts entered into the system by this user.
- The receipts this user has been added to as a debtor.
- The payments this user still has to make to other users.
- The payments this user still has to receive from other users.

The information on the payments the users still has to make and receive should take all Receipts and Payments entered into the system into account.

If the page belongs to the currently logged in user, then the user should be presented with the option of entering new payments into the system.

`/receipt` Should list all receipts that have been entered into the system, linking to their `/receipt/`*`receiptId`* page.

`/receipt/`*`receiptId`* Should display information about the given receipt (the user who entered it into the system, the amount, and the users who should pay a part of the amount to the one that entered the receipt into the system).

`/payment` Should display all payments that have been entered into the system.

Furthermore, you may assume that:

1. Only one User every pays a bill and that this is always the User that enters the Receipt into the system.
2. The costs of a bill are always split evenly between all debtors.

Optional requirements

Here are some suggested features you can try to implement:

Additional entities The minimal requirements make some assumptions that are not always realistic: for example, it assumes the bill is always split equally and that only one person paid for the bill. Lift these restrictions. This will probably require you to modify several of the entities and/or add new ones.

Update and delete The minimal requirements only state you should be able to Create and Retrieve entities. Modify you application so that it can also Update (edit, modify) and Delete the various entities.

Validation Make sure all user input gets properly validated. For example, users probably shouldn't be able to enter receipts or payments for negative amounts into the system. (See: "Forms: Validation".)

Navigation bar Modify the `defaultLayout` function, so that a navigation bar is automatically displayed on all pages, allowing easy access to the various pages in your application. (See: "Yesod Typeclass: defaultLayout".)

Sorting Make some of the tables you output sortable on a user selected column. (See: "Yesod's Monads: Example: Request Information".)

Authorization The starting framework already handles *authentication* for you: a visitor of the website can login and is then associated with a User entity in the database. However, no *authorization* is happening: anyone can see any page in the system. Add some authorization checks to the application. (See: "Authentication and Authorization".)

Testing Write QuickCheck tests for the pure portions of your code, or use the `Yesod.Test` framework for testing the impure portions of your application. (See the `/tests` folder in the starting framework.)

Machine readability Make the pages accessible in a machine readable form as well (for example, in the JSON format) and ideally through the same URL. (See: "RESTful Content: Representations".)

If you want to add other features to your application this is fine to. However, we are mainly interested in seeing how well you are at writing Haskell, so any features that consist purely of HTML, CSS or JavaScript code will only count minimally towards your grade.

5.2.4 Starting framework

Overview of files

`config/models` In this file you must define the entities that are stored in the database. A beginning has been made here, but you will have to fill out the rest. Note that if you make changes to the database schema then Yesod will try to migrate your database to the new schema automatically. If the changes are too complex then this will fail and you will have to delete the database (`IOU.sqlite3`) manually and start with a fresh one.

`config/routes` In this file you define the URLs of the pages the web application responds to. Several pages needed to meet the minimal requirements are present already, but you may need to add more.

`Handler/*.hs` These modules make up the core of the web application and take care of processing user input, querying the database, and rendering the output. Most functions have been left undefined, so here you will be doing most of your work. If you want to add extra modules here, then don't forget to list them in `Application.hs` and `IOU.cabal` as well.

`messages/en.msg` Here you can define messages, if you want to make use of the internationalization framework (optional).

5.2.5 Hints

Write functional code

Most of the code in the `Handler/*.hs` files will run inside the `Handler Html` monad that allows you to retrieve user input, query the database, and eventually return some HTML.

However, a lot of the processing that needs to be done between retrieving the user input and query the database, and between querying the database and returning the resulting web page can be done functionally. Try to define helper functions that do not run inside the `Handler` monad in order to do this processing purely functionally.

Many-to-many relations

The User and Receipt entities have a many-to-many relationship with one another: a Receipt can list many Users as debtors, and a User can be listed as a debtor on many Receipts.

Modeling such a relation in a relational database requires you to add an extra ReceiptUser entity matching Users to Receipts. Performing a query on such relations requires you to perform a two or three-way join on those relations. For this purpose the `joinTable` and `joinTable3` are available. See `getPaymentsR` in `Handler/Payment.hs` for an example of how to use the latter function.

Users

You can use a Google account to log in to the application (your `@students.uu.nl` address should be associated with such an account). However, for testing purposes it is convenient to have a few extra Users present in the system. So unless you have a few spare Google accounts laying around, you might want to `insert` a few extra users into the database.

6 Introduction to Agda

To complete these exercises, you will need to install the dependently typed programming language Agda. You can find the most recent installation instructions on the Agda wiki. You may also want to consult the list of most common Agda commands and Emacs cheatsheet.

You can download a template Agda file from the AFP website, containing type signatures and holes. Your job is to complete the remaining definitions.

Exercise 6.0.1. Show that the type `Vec a n` is an applicative functor in `a`, that is, define the following functions:

```
pure : {n : Nat} {a : Set} -> a -> Vec a n

_<*>_ : {a b : Set} {n : Nat} -> Vec (a -> b) n -> Vec a n -> Vec b n

vmap : {a b : Set} {n : Nat} -> (a -> b) -> Vec a n -> Vec b n
```

Exercise 6.0.2. Besides one dimensional vectors, we can also define a type of *matrices*, that is vectors of vectors. Show how to add two matrices, multiply to matrices, create the identity matrix, and transpose a matrix. You may find the applicative definitions from the previous exercise useful.

```
Matrix : Set -> Nat -> Nat -> Set
Matrix a n m = Vec (Vec a n) m

madd : {n m : Nat} -> Matrix Nat m n -> Matrix Nat m n -> Matrix Nat m n

mmul : {n m k : Nat} -> Matrix Nat m n -> Matrix Nat n k -> Matrix Nat m k

idMatrix : {n : Nat} -> Matrix Nat n n

transpose : {n m : Nat} {a : Set} -> Matrix a m n -> Matrix a n m
```

Exercise 6.0.3. The data type `Fin n` can be used to represent a type with precisely `n` inhabitants.

```
  data Fin : Nat -> Set where
```

```
    Zero : forall {n} -> Fin (Succ n)
    Succ : forall {n} -> Fin n -> Fin (Succ n)
```

Show how to create a vector of length `n`, enumerating all the inhabitants of `Fin n`:

```
plan : {n : Nat} -> Vec (Fin n) n
```

Next, define a simple forgetful map from `Fin n` to `Nat`:

```
forget : {n : Nat} -> Fin n -> Nat
```

There are several ways to embed

There are several ways to embed `Fin n` in `Fin (Succ n)`. Try to come up with one that satisfies the correctness property below (and prove that it does).

```
embed : {n : Nat} -> Fin n -> Fin (Succ n)

correct : {n : Nat} -> (i : Fin n) -> forget i == forget (embed i)
```

Exercise 6.0.4. The next exercise is to define several functions for comparing natural numbers. Given the following data type:

```
  data Compare : Nat -> Nat -> Set where
    LessThan : forall {n} k -> Compare n (n + Succ k)
    Equal : forall {n} -> Compare n n
    GreaterThan : forall {n} k -> Compare (n + Succ k) n
```

Show that there is a 'covering function' – that is, that for any pair of numbers, you can always decide which one is biggest:

```
cmp : (n m : Nat) -> Compare n m
```

Finally, use this comparison function to define the absolute difference between two numbers:

```
difference : (n m : Nat) -> Nat
```

Exercise 6.0.5. Prove the following lemmas formulated below. You may want to define auxiliary lemmas or use the notation intoduced in the lectures.

```
plusZero : (n : Nat) -> (n + 0) == n

plusSucc : (n m : Nat) -> Succ (n + m) == (n + Succ m)

plusCommutes : (n m : Nat) -> (n + m) == (m + n)

distributivity : (n m k : Nat) -> (n * (m + k)) == ((n * m) + (n * k))
```

Exercise 6.0.6. Consider the following relation between two lists, describing when one list is a sublist of another:

```
data SubList {a : Set} : List a -> List a -> Set where
  Base : SubList Nil Nil
  Keep : forall {x xs ys} -> SubList xs ys -> SubList (Cons x xs) (Cons x ys)
  Drop : forall {y zs ys} -> SubList zs ys -> SubList zs (Cons y ys)
```

Show that this relation is an equivalence relation, i.e., prove the following three properties:

```
SubListRefl : {a : Set} {xs : List a} -> SubList xs xs

SubListTrans : {a : Set} {xs ys zs : List a} ->
  SubList xs ys -> SubList ys zs -> SubList xs zs

SubListAntiSym : {a : Set} {xs ys : List a} ->
  SubList xs ys -> SubList ys xs -> xs == ys
```

Exercise 6.0.7. Define a relation between two natural numbers, stating that one is less-than-or-equal to the other:

```
data LEQ : Nat -> Nat -> Set where
  ...
```

Prove that your relation is also an equivalence relation:

```
leqRefl : (n : Nat) -> LEQ n n

leqTrans : {n m k : Nat} -> LEQ n m -> LEQ m k -> LEQ n k

leqAntiSym : {n m : Nat} -> LEQ n m -> LEQ m n -> n == m
```

Now given the following check that one number is less than the other:

```
_<=_ : Nat -> Nat -> Bool
Zero <= y = True
Succ x <= Zero = False
Succ x <= Succ y = x <= y
```

Show that your data type is inhabited precisely when this function returns true.

```
leq<= : {n m : Nat} -> LEQ n m -> (n <= m) == True

<=leq : (n m : Nat) -> (n <= m) == True -> LEQ n m
```

Exercise 6.0.8. We can define logical negation as follows:

```
Not : Set -> Set
Not P = P -> Empty
```

Agda's logic is *constructive*, meaning some properties you may be familiar with from classical logic do not hold. Prove the following property:

```
notNotP : {P : Set} -> P -> Not (Not P)
```

The converse implication does not hold: `Not (Not P)` does not imply `P`.

Similarly, `P or Not P` doesn't hold for all statements `P`, but we can prove the statement below. It's an amusing brainteaser.

```
data Or (a b : Set) : Set where
  Inl : a -> Or a b
  Inr : b -> Or a b

orCase : {a b c : Set} -> (a -> c) -> (b -> c) -> Or a b -> c
orCase f g (Inl x) = f x
orCase f g (Inr x) = g x

notNotExcludedMiddle : {P : Set} -> Not (Not (Or P (Not P)))
notNotExcludedMiddle = {!!}
```

There are various different axioms that can be added to a constructive logic to get the more familiar classical logic.

```
doubleNegation = {P : Set} -> Not (Not P) -> P
excludedMiddle = {P : Set} -> Or P (Not P)
impliesToOr = {P Q : Set} -> (P -> Q) -> Or (Not P) Q
```

Prove these three statements are equivalent. You may find it helpful to replace the definitions such as `doubleNegation` their definition in the type signatures below.

```
step1 : doubleNegation -> excludedMiddle

step2 : excludedMiddle -> impliesToOr

step3 : impliesToOr -> doubleNegation
```

A harder challenge is to show that these are also equivalent to Pierce's law:

```
piercesLaw = {P Q : Set} -> ((P -> Q) -> P) -> P
```

Exercise 6.0.9. In this final exercise you will verify a tiny compiler.

The following data types for expressions constitutes our source language:

```
data Expr : Set where
  Val : Nat -> Expr
  Add : Expr -> Expr -> Expr
```

Next, define an evaluation function, that computes the natural number to which an argument expression will evaluate.

```
eval : Expr -> Nat
```

Our target language will consist of the following instructions:

- `PUSH n` will push the natural number `n` on our stack;
- `ADD` will take two numbers from the stack and add them.

Define a data type representing this target language.

Show how to execute these instructions:

```
exec : Instructions -> List Nat -> List Nat
```

You cannot give a total definition with the desired semantics. You can fix this by returning a value of type `Maybe (List Nat)` instead; a more elegant fix, however, is to add more (dependent) types describing the size of the stack.

Finally define a compiler:

```
compile : Expr -> Instructions
```

And prove that for every expression, evaluation produces the same result as executing the corresponding compiled code.