The guide to

Functional Programming

λ

Michael B. Gale
m.gale@warwick.ac.uk

2020/21

Contents

λ.1

THE MODULE

Functional Programming is an optional module which follows on from introductory programming modules such as CS118 or equivalents in other departments. In such modules you typically learn to write programs in the imperative style in languages such as Java, C, or Python. However, there are many different programming languages and many different programming paradigms. The imperative and object-oriented programming paradigms that you have learnt so far are just two of them. You can think of programming languages as tools: a hammer is different from a screwdriver and both serve different purposes for which they are the right choice of tooling. The same is true for programming languages. It is easier or harder to accomplish certain tasks in some languages than it is in others. To be a good programmer, you need to know which tools are at your disposal and when to use them.

In this module, you will learn about the *functional programming* paradigm, which is equally as important as imperative and object-oriented programming. No prior programming knowledge is required for this module and it is therefore suitable for most scientists. We use the *Haskell* programming language in this module. It is one of many functional programming languages, but it is quite unique in that it is lazy and purely functional. Writing programs in functional languages, and particularly Haskell, is very different from writing programs in languages like Java. Over the course of this module you will learn how to do that. This adds a powerful tool to your programming arsenal and you will gain a much deeper understanding of programming as a whole. Skills from this module can be applied in other languages, functional or not. In other words, you will become a better programmer!

This guide serves as a companion to the module by giving you an overview of all the major components, including guidance on how to use the different tools you will encounter as part of this module. You can also find the coursework specifications as well as exercises for all of the labs in this guide.

1.1 Books

You are not required to purchase any books for this module as there are many resources available online, including a number of free books. For those of you who prefer to have a book with most of the content in one place, I can recommend the following, all of which are introductory texts. Each book which is recommended here teaches functional programming in a different way[1]:

Learn you a Haskell for Great Good!
Miran Lipovača, No Starch Press.

This book is also a general introduction to Haskell with a greater emphasis on concepts in functional programming. If you prefer to learn by focusing on theory and concepts which you can then later apply to problems when you encounter them, this book is for you. It is also available for free on the book's website at:

`http://learnyouahaskell.com/`

Programming in Haskell (2nd edition)
Graham Hutton, Cambridge University Press.

A general introduction to Haskell which is well-structured and covers all of the major topics from the lectures in a vaguely similar order. This is the "main text" and this guide includes references to chapters in Hutton's book for further reading. Note: if you are thinking of buying the first edition, which is likely cheaper at this point, it doesn't cover the later, more advanced topics of this module.

Real World Haskell
Bryan O'Sullivan, Don Stewart, and John Goerzen, O'Reilly.

This book focuses heavily on solving practical tasks using Haskell after only a brief introduction to the basic concepts. This book is for you if you prefer to learn by seeing how particular techniques are used in action. This book is quite old now, so some of the libraries and techniques used in the book may be outdated by now, but the book is also available for free on the book's website at:

`http://book.realworldhaskell.org/`

[1]This reading list is also available on the library website at `https://warwick.rl.talis.com/modules/cs141.html` which lets you find copies in the library.

The Haskell School of Expression
Paul Hudak, Cambridge University Press.

This book teaches functional programming graphically at the start and later through music. If you like visual and auditory results while you are learning, this book may be for you. Like Real World Haskell, this book is slightly older and some of the code shown may not work without modifications.

Introduction to Functional Programming
Richard Bird and Philip Wadler, Prentice Hall.

This is an excellent, but old, introduction to functional programming. This book came out before I was born and uses a Haskell precursor language called Miranda. However, it does a very good job at teaching the foundations of functional programming.

1.2 Timeline

This module is comprised of approximately 30 lectures, 10 labs, 2 pieces of coursework, and an exam. This section contains a chronological schedule of all of these components. Note that the schedule may be subject to changes due to *e.g.* staff illness or other unforeseen circumstances. Each lecture aims to answer a specific question, which is shown in the timeline. You can test your understanding by asking yourself that question after each lecture and checking that you can answer it.

The definite timetable is available on Tabula[2], the module website, or on your personalised timetable on Tabula as well.

11 January

Lecture 1: Introduction

What is functional programming and why should we learn it?

Overview of programming paradigms & models of computation, examples of applications of functional programming, module overview, and recommended texts.

12 January

Lecture 2: Definitions & functions

How do we write simple programs in Haskell?

Definitions, basic arithmetic expressions, string values, boolean values, functions, using built-in functions, and basic pattern matching.

[2]`https://tabula.warwick.ac.uk/profiles/department/cs/timetables?modules=cs141`

13 January

Lecture 3: Basic types

How does the compiler prevent us from writing bad software?

Basic types, function types, parametric polymorphism, constraints, and pairs.

11-15 January

Week 1 exercises

Relevant exercises for this week are *Getting started*, *Functions*, and *Basic types*.

18 January

Lecture 4: Lists

How do we use lists in Haskell?

Constructing lists, pattern-matching on lists, and list comprehensions.

19 January

Lecture 5: Type classes

How can we restrict polymorphism and overload functions?

Ad-hoc polymorphism via type classes, built-in type classes, and type class constraints.

20 January

Lecture 6: Recursive functions

How do we express loops without mutable state?

Writing recursive functions for basic types (numbers, lists/strings), defining built-in functions ourselves.

18-22 January

Week 2 exercises

Relevant exercises for this week are *Using the standard library*, *Lists*, *List comprehensions*, *Type classes*, and *Recursive functions*.

25 January

Lecture 7: Data types & type aliases

How can we define our own types in Haskell?

Type aliases, data types, data constructors, pattern matching on custom data constructors, recursion on values of custom types.

26 January

Lecture 8: Coursework I briefing

What is the first coursework about?

Demonstration of what the completed coursework will do, explanation of the rules, introduction to the skeleton code.

27 January

Lecture 9: Higher-order functions

Can we write functions which abstract over common behaviours?

Higher-order functions such as `map`, `filter`, etc., recursion primitives such as `foldr` and `foldl`.

25-29 January

Week 3 exercises

Relevant exercises for this week are *Data types*, *Higher-order functions*, and *Using other libraries*.

1 February

Lecture 10: Higher-order functions (cont.)

Can we write functions which abstract over common behaviours?

Higher-order functions such as `map`, `filter`, etc., recursion primitives such as `foldr` and `foldl`.

2 February

Lecture 11: Lazy evaluation

What order are programs evaluated in?

Strict and lazy evaluation, calling conventions, benefits and disadvantages, and examples of lazy evaluation.

3 February

Lecture 12: Testing

What tools are there for testing and how do we use them?

Unit testing, property-based testing, and code coverage in Haskell.

1-5 February

Week 4 exercises

Relevant exercises for this week are *Higher-order functions* and *Lazy evaluation*.

8 February

Lecture 13: Reasoning about programs

Can we use formal reasoning techniques to prove properties about our programs?

Equational reasoning, proofs by induction.

9 February

Lecture 14: Reasoning about programs (cont.)

Can we use formal reasoning techniques to prove properties about our programs?

Equational reasoning, proofs by induction.

10 February	**Lecture 15: Constructive induction** *Can we use formal reasoning techniques to calculate more efficient programs?* Using induction to calculate faster functions.
8-12 February	**Week 5 exercises** Recommended exercises for this week are *Proofs*.
16 February	**Deadline: Coursework I**
15 February	**Lecture 16: Foldables & Functors** *Are there any useful design patterns in functional programming?* `Foldable` and `Functor` type classes, their motivation, and examples.
16 February	**Lecture 17: Coursework II briefing** *What is the second coursework about?* Demonstration of what the completed coursework will do, semantics of the programming language, introduction to the skeleton code.
17 February	**Lecture 18: Introduction to Applicative Functors** *Are there any other useful design patterns in functional programming?* Applicative functors.
15-19 February	**Week 6 exercises** Relevant exercises for this week are *Foldables* and *Functors*.
22 February	**Lecture 19: Applicative functors** *What can we do with applicative functors?* Applicative functors, applications of applicative functors.
23 February	**Lecture 20: Applicative functors (cont.)** *What can we do with applicative functors?* Applicative functors, applications of applicative functors.

24 February

Lecture 21: Introduction to sequential composition

How do structure programs in which one part of a program relies on the result of another part?

Some functions for the sequential composition of `Maybe` values.

22-26 February

Week 7 exercises

Relevant exercises for this week are *Applicative functors*.

1 March

Lecture 22: Sequential composition

Are there other examples of sequential composition?

Some functions for the sequential composition of `State` and `Writer` values and some laws for sequential composition.

2 March

Lecture 23: Sequential composition (cont.)

Are there other examples of sequential composition?

Some functions for the sequential composition of `State` and `Writer` values and some laws for sequential composition.

3 March

Lecture 24: Input and output

Can we write impure programs in a pure programming language?

The `IO` monad.

1-5 March

Week 8 exercises

Relevant exercises for this week are *Effectful programming*.

8 March

Lecture 25: Writing a real application in Haskell

What do real Haskell programs look like?

Everything we have learnt so far in action.

9 March

Lecture 26: Writing a real application in Haskell (cont.)

What do real Haskell programs look like?

Everything we have learnt so far in action.

10 March

Lecture 27: Type promotion & GADTs

How can we encode more information in types?

Phantom types, GADTs, singleton types, pattern matching with GADTs.

8-12 March	**Week 9 exercises** Relevant exercises for this week are *Input & output* and *Kinds*.
15 March	**Lecture 28: Type families** *How can we perform computation at the type-level?* Closed and open type families.
16 March	**Lecture 29: Applied type-level programming** *What are some examples of how type-level programming is used?* Type-level programming in action.
17 March	**Lecture 30: Conclusions** *What have we learnt about functional programming?* Summary of the module, information about the exam, and other general information.
15-19 March	**Week 10 exercises** Relevant exercises for this week are *Type-level programming*.
23 March	**Deadline: Coursework II**
Term 3	**Revision lectures** Student-selected topics from the previous lectures.
Term 3	**Exam** 2 hours. Answer any four out of six questions.

1.3 Coursework (15% + 25%)

There are two pieces of coursework which you will have to complete during Term 2 and submit through Tabula. You will receive feedback for the first coursework before the second coursework is due. Once you have made a submission through Tabula, it will be processed by our custom software called WAAT. After submission, you will have access to WAAT at

```
https://waat.dcs.warwick.ac.uk/
```

where you can view all your submissions and the reports that our system has automatically derived from them. You can use this to ensure that your submission compiles and works properly on our systems.

1.3.1 Large Arithmetic Collider (15%)

Description You have to implement a program in Haskell which can solve a challenging combinatorial problem which we have named the *large arithmetic collider*.

Aims This coursework is designed to test your ability to write basic Haskell programs using built-in functions, work with lists, and write recursive functions.

1.3.2 Scratch clone (25%)

Description Scratch is a popular tool for teaching programming to children. For this coursework, you have to implement an interpreter for a simple programming language which is used to complete a clone of Scratch.

Aims This tests your ability to make use of type classes, data types, and functional design patterns such as monads.

1.4 Exam (60%)

The exam is worth 60% of the module and takes place in Term 3. You will have to answer any four out of six questions. Each question is worth 25 marks. Past exam papers as well as a sample exam paper are available on the module website. Note that since the exam in 2020/21 will be online, the format of some questions may change accordingly and an updated sample paper will be made available reflecting this. In addition to the past papers on the module website, you may also be able to find CS256 exam papers from before 2017/18, but note that their format and content are significantly different and I would not recommend those for revision.

In general, I prefer to set exam questions which require you to use your understanding of functional programming and Haskell to solve problems of varying difficulties. There are unlikely to be any bookwork questions and there will be no lengthy essay questions for you to answer. A reference of the Haskell standard library which includes the types and simplified definitions of many useful functions may be found in Chapter 9.

There will be at least three revision lectures in Term 3, the dates of which are to be confirmed. I typically discuss some tips and tricks that allow you to answer questions more easily, so you are strongly encouraged to attend. You are also welcome to suggest topics if you wish.

λ.2
TOOLS

Good programmers are lazy. That is because laziness encourages us to seek out the simplest solutions to our problems. Often, this will be achieved by using suitable programming abstractions to write reusable and concise code. However, in order to focus on actually writing code, it is important that we have the right tools to support us in doing so.

There are a number of programs we use as part of this module. Some are optional and simply make your life easier, while others are essential. You will need to familiarise yourself with them in order to complete the labs and coursework successfully. This section contains an overview of all programs we use or recommend you use with short summaries of what each program does. There are also instructions on how to get started either using the machines in the department or your own.

2.1 Getting started on the departmental machines

All the tools we need for this module are already installed on the machines in the department. However, the only thing you need to do to get started is run `/modules/cs141/haskell-setup.sh` in a terminal which will set up a number of things for you:

- It will make the `stack` tool work on your account (see Section 2.3.3 for what `stack` is).
- It will install a number of VSCode plugins (see Section 2.5) for you which will help you write Haskell programs in VSCode.

You should do this before you do anything else and you only need to do this once.

2.2 Getting started on your own computer

If you also want to work on your own machine, there are several ways in which you may install a Haskell distribution.

2.2.1 Option 1: Installation using Stack (recommended)

Download the Haskell Stack from `https://docs.haskellstack.org/en/stable/README/`. There is a simple command you can run on Unix-based machines and there are installers for Windows. You can also get it from your system's package manager. Once you have installed Stack, you can install the Haskell compiler, GHC, using it as follows. For best compatibility with the module and to save you time, you are encouraged to specify the resolver version that we use for the module this year:

```
$ stack setup --resolver=lts-16.27
```

That may take a few minutes to complete, but this is all you need to do.

2.2.2 Option 2: Haskell Platform

You can download the Haskell Platform from the `https://www.haskell.org/` website or from a package manager of choice (beware of old versions!). The Haskell Platform contains the Haskell compiler, GHC, as well as a comprehensive set of libraries. The downside to taking this approach is that the libraries which are contained in a release of the Haskell Platform may not be the most recent versions or you may not need all of them to begin with. Note that libraries can be installed on-the-fly at any time anyway, so there is no need to have a large set of them pre-installed. Also, you will still want to install Stack to help you with building, testing, and benchmarking practicals and coursework.

2.2.3 Option 3: GHC only

If you are feeling really adventurous, you can install just GHC from the GHC website. You will then have to add any tools or libraries you want manually.

2.3 Haskell

Haskell is a modern, functional programming language and the primary language we use in this module. The Haskell language has been around for over twenty years and has evolved throughout that time. You can find more information about Haskell on the following website:

`https://www.haskell.org/`

Like with most other programming languages, Haskell source files are just plain text files. They are conventionally given the `.hs` file extension so that we can identify them more easily. In order to compile a Haskell source file, you of course need a Haskell compiler.

2.3.1 GHC

The Glasgow Haskell Compiler (GHC) is the most mature and feature-rich Haskell compiler out there and is the compiler we use in this module. Its implementation of Haskell is the *de facto* language standard as far as many people are concerned. In addition to the core Haskell language, it also includes many extensions to the language which reflect state-of-the-art programming language features that stem from current research in the field. You can find more information about GHC at:

`https://www.haskell.org/ghc/`

GHC is already installed on the department's computers. You can invoke the version that `stack` installed simply by running `stack ghc` in a terminal window. For example, suppose that we have a file named `Program.hs` with the following contents:

```haskell
main = putStrLn "Hello World!"
```

In order to compile this program, you could run the following command in a terminal:

```
$ stack ghc Program.hs
```

This would produce (among other things) an executable named `Program` that can then be run to produce the expected output:

```
$ ./Program
Hello World!
```

2.3.2 GHCi

GHC typically compiles Haskell source files into executables or libraries. However, it can also be used *interactively* to provide you with a text-based user interface which lets you evaluate expressions that you type in, experiment with functions you have defined, etc. In this mode, GHC is referred to as GHCi, or GHC interactive. GHCi can be invoked by running `stack ghci` in a terminal window. You can optionally specify any Haskell source files you wish to load as additional arguments.

2.3.3 Haskell Stack

Haskell programs, like programs written in other languages, typically consist of more than just one source file, may depend on libraries which provide functionality that we do not wish to implement from scratch, have test suites, benchmarks, and so on. Haskell Stack is a *build tool* which automates many of tasks related to GHC, such as downloading and installing different versions of GHC, managing and building projects, managing dependencies, running unit tests, running benchmarks, etc. You can find more information about Stack at:

`https://www.haskellstack.org`

Stack is already installed on the departmental machines, but you need to run the following command in a terminal window to set it up on your user account if you have not yet done so:

```
$ /modules/cs141/haskell-setup.sh
```

For Stack to work with a particular project, it needs a configuration file. All exercises for the labs and the coursework already come with a Stack configuration file, so you do not have to configure anything yourself. These files are named `stack.yaml` and you can find them in the root folders of each lab or coursework.

Once you have obtained *e.g.* the skeleton code for one of the labs, you can run the following command to compile the code in the folder that contains the skeleton code:

```
$ stack build
```

This will invoke GHC to compile all of the source files and also link all dependencies specified in the project's `.cabal` file into the program. Any errors that occur during compilation will be reported to the standard output. Many of the labs and all of the coursework will also come with a test suite. You can run the test suite by invoking:

```
$ stack test
```

As the test suite is being executed, it will print the outcome of each test to the standard output. If there are any benchmarks, you run them by invoking:

```
$ stack bench
```

Another useful command is the following, which invokes GHCi with the configuration from the `stack.yaml` file:

```
$ stack repl
```

or equivalently:

```
$ stack ghci
```

This launches the GHCi REPL for your project so that you can experiment with your code and ask for types etc. See the notes for the first exercises for details on the REPL.

2.3.4 Cabal

Cabal is Haskell's default package manager and build tool. We will not be using it in this module since Stack does everything Cabal does. However, you should be aware of files with the `.cabal` extension which contain the project configuration, such as which dependencies to load. Stack can also use a tool called `hpack` to generate these `.cabal` files from `package.yaml` files. Some labs will just come with a `.cabal` file which can you edit directly while others may have a `package.yaml` file that you need to edit instead. If you decide to use *e.g.* additional libraries (Section 2.3.6), then you will need to list them in the relevant `.cabal` or `package.yaml` file. We cover this process in detail in one of the lectures.

2.3.5 Prelude

The `Prelude` is Haskell's standard library. It is part of the `base` package. It contains many useful types and functions which you will make use of in virtually every program. The `Prelude` module is automatically imported into every Haskell module and the `base` package is automatically imported into every Haskell project, so you do not have to do anything to use it. You can find documentation for all the functions and types offered by the `Prelude` at:

`http://hackage.haskell.org/package/base/docs/Prelude.html`

2.3.6 Hackage

The `Prelude` is of course not the only library that is available for Haskell. There are many different libraries which offer a lot of useful functionality. Hackage is Haskell's package database. If you are looking for a library which *e.g.* provides a particular data structure, you can look on `Hackage` for it:

`http://hackage.haskell.org/`

Note: if you are using the departmental machines, you will not be able to install any libraries off Hackage using `stack`, but you can use the ones which I have installed. If you would like to use a library on the departmental machines which is not installed, please let me know and I can install it for you!

If you are working on your own machine and want to install additional packages, you can do so with Stack. For example, if you want to install the `containers` package, you can run `stack install containers`. Make sure to do this in the folder which contains the project you want to use `containers` with as Stack will only install it for that project. See the Stack documentation for more details.

2.4 Version control

The skeleton code for all practicals and for all coursework is available as Git repositories which are hosted on GitHub at `https://github.com/fpclass/`. GitHub is one of several web services that allows you to host Git repositories online, along with *e.g.* GitLab and BitBucket which are also popular. You are encouraged to use version control to obtain and maintain your code. If you have not had much exposure to version control using Git before, *Pro Git* by Scott Chacon and Ben Straub is a very good reference book which is available for free at `https://git-scm.com/book/en/v2`. Below is a very quick reference of some of the most important commands you will use.

To obtain the code for e.g. the first practical which is located in the `lab-getting-started` repository, you will want to run the following command on your machine once you have installed `git` (note that `git` is already installed by default on the lab machines, macOS, and many linux distributions):

```
$ git clone https://github.com/fpclass/lab-getting-started
```

This will create a local copy of the `lab-getting-started` repository on your machine that you can modify. If you are planning to work on your practical or coursework from multiple locations (e.g. a machine in the lab and your personal machine), you may find it beneficial to create a GitHub account and *fork* the relevant repositories to your account instead. This creates a copy of them on your GitHub account which can then be read from and written to from anywhere. You can fork repositories on the GitHub website by visiting e.g. `https://github.com/fpclass/lab-getting-started` once logged in and clicking the "fork" button. If you take this approach, you will still need to obtain a local copy of the repository on all machines you plan to work on by running the command shown above, but replacing `fpclass` with your GitHub username.

WARNING: do not fork the coursework repositories to your account as they will end up being public and you do not want everyone to be able to see your solutions! Use the GitHub Classroom links provided on the module website or at the start of each coursework specification instead which will allow you to create *private* forks of the repositories.

Once you have made some changes to the skeleton code, you will want to *commit* your changes. This will tell `git` to remember that version in case you ever wish to go back to it. You can commit your changes by running:

```
$ git commit -m 'Some message to describe the changes' -a
```

If you forked the repository to your own GitHub account, you may now wish to update that repository with your local changes by running:

```
$ git push
```

This will update the repository on GitHub with all changes you have made. You may then wish to run the following command if you continue working on another machine, which will update the local repository there with changes from GitHub:

```
$ git pull
```

None of this is necessary if you have cloned the code from `https://github.com/fpclass/` directly, but then you also cannot work on the code from multiple machines, unless you store the local repository on *e.g.* Dropbox or a similar service (which is not recommended).

2.5 Development tools

Most IDEs and text editors have plugins which can help you write Haskell code, by adding syntax highlighting or other useful features. More sophisticated plugins require additional tools which provide programming language-specific functionality to the editor.

2.5.1 ghcid

A lightweight development tool for Haskell is `ghcid`, which is essentially just `ghci` or `stack repl`, but automatically reloads your files when they change. You can read more about `ghcid` at:

`https://github.com/ndmitchell/ghcid`

The `ghcid` executable is already installed on the lab machines and you can invoke it by running the following in *e.g.* a folder with a `stack.yaml` file in it:

```
$ /modules/cs141/bin/ghcid
```

The program will continue to run in the background and, every time you change any files in your project, compile them automatically. If any errors arise, the tool will output them.

2.5.2 Haskell Language Server

Haskell Language Server (or HLS for short) is the Haskell community's current effort at creating a unified tool that text editors and IDEs can use to provide Haskell-related functionality. The `haskell-language-server` program implements Microsoft's Language Server Protocol (LSP) which allows HLS to be used with any editor that implements LSP. The tool does not currently work on the lab machines, but if you are working on your own machine, you can install it by following the instructions at:

`https://github.com/haskell/haskell-language-server`

2.5.3 Editors

This section contains recommendations for Haskell-related plugins for different text editors.

Visual Studio Code Visual Studio Code is the text editor I would recommend and that I am currently using. The `/modules/cs141/haskell-setup.sh` script on the departmental machines already installs the following plugin for you:

- `language-haskell`, which provides syntax highlighting for `.hs` source files.

On your own machine, I would additionally recommend the `haskell` plugin which allows VSCode to provide IDE-like functionality for Haskell by utilising `haskell-language-server`.

Atom Atom is another text editor similar to VSCode that I can recommend for writing Haskell programs. There are a number of Haskell-related plugins which you may wish to install:

- `language-haskell`, which provides syntax highlighting for `.hs` source files.
- `atom-ide-ui`, which provides IDE-like UI elements in Atom.

- `haskell`, which allows Atom to provide IDE-like functionality for Haskell by utilising `haskell-language-server`.

If you are using Atom on your own machine, I would strongly encourage you to install the above packages through Atom's package manager `apm`.

Sublime text Sublime is a commercial text editor and that you need to pay for, but it offers better performance than Atom or VSCode. If you own a copy of Sublime and want to use `haskell-language-server` with it, you can follow the instructions at:

`https://github.com/haskell/haskell-language-server#using-haskell-language-server-with-sublime-text`

Vim To use `haskell-language-server` with Vim, you can follow the instructions at:

`https://github.com/haskell/haskell-language-server#using-haskell-language-server-with-vim-or-neovim`

There are also Vimscripts for Haskell at:

`https://github.com/neovimhaskell/haskell-vim`

Emacs To use `haskell-language-server` with Emacs, you can follow the instructions at:

`https://github.com/haskell/haskell-language-server#using-haskell-language-server-with-emacs`

There is a Haskell mode for Emacs as well:

`https://github.com/haskell/haskell-mode`

2.6 Other useful resources

2.6.1 The lecturer & lab tutors

Obviously. We are happy to help! Feel free to get in touch on Slack at any time (for usually pretty quick responses), or send me an email at m.gale@warwick.ac.uk if you have any questions (for slightly slower responses).

2.6.2 Hoogle

If you ever need to find something in a library, Hoogle[1] is an incredibly valuable resource. I use it all the time! You can use it for a range of different things:

- If you know the name of a function and want to find out more about it or what its type is, you can just search for the name. The results will take you to the documentation for the relevant module.
- If you want to find a function which does something in particular, *e.g.* applies a function to all elements of a list, you can search for the type and Hoogle will try to find a matching function for you. It will automatically rearrange parameters and search for similar functions or those with more general types as well.

2.6.3 Reddit

There is a Haskell subreddit[2] where people often post Haskell-related news and discussions. If you use Reddit regularly (*i.e.* too much) and want to broaden your Haskell horizons, it might be a good idea to subscribe.

2.6.4 The Haskell mailing list

There are several Haskell mailing lists[3]. The most interesting ones for you will be `Haskell-Cafe` and `Beginners`. The latter is for beginner questions. This might be a good place to get help from if you are thinking of using Haskell after the course has finished. Until then, you are encouraged to seek help from me or one of the tutors instead – we are happy to help!

2.6.5 Other communities

There are plenty of other Haskell communities on the web[4] you can join and take part in. Remember, you cannot ask other people to solve all or parts of your coursework for you. We will find out – we have the technology!

[1]`https://hoogle.haskell.org`
[2]`https://www.reddit.com/r/haskell/`
[3]`https://www.haskell.org/mailing-lists`
[4]`https://www.haskell.org/community`

λ.3 Lecture notes

This chapter contains some lecture notes for selected topics from the lectures for which there are not necessarily many resources available elsewhere.

3.1 Introduction to equational reasoning

3.1.1 Natural numbers

Suppose that we define natural numbers as previously shown in the lecture on algebraic data types. That is, a natural number is either zero, represented by the Z constructor, or the successor to some other natural number, represented by the S constructor which takes a natural number as argument:

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

We can then define addition as a recursive function:

$$\begin{array}{llcl}
\multicolumn{4}{l}{add :: Nat \to Nat \to Nat} \\
add\ \ Z & m & = & m \\
add\ \ (S\ n) & m & = & S\ (add\ n\ m)
\end{array}$$

Left identity There are some well-known properties of addition. For example, addition has a left unit or identity: adding zero to some number m should just return m:

$$\forall m :: Nat\ .\quad add\ Z\ m = m$$

The proof for this equality is trivial, since it is an exact match of one of the equations which define *add*. We write down this proof as follows:

$$\begin{aligned} & add\ Z\ m \\ = & \quad \{ \quad \text{applying } add \quad \} \\ & m \end{aligned}$$

In this case, we started with the left-hand side of the equation and justified that we can just rewrite it to the right-hand side by the definition of *add*. We will continue to use this format for all of our future proofs.

Right identity Similarly to the first property, adding some natural number n to zero should also be equivalent to just n:

$$\forall n :: Nat\ .\quad add\ n\ Z = n$$

However, this time we cannot just use the definition of *add* to rewrite one side of the equation to the other, because the result of *add* depends on what the first argument is. Since we do not know what n is, we must explore all possible options through induction on n. In the base case, where $n = Z$, we can show that $add\ Z\ Z$ is equivalent to Z just by rewriting one side of the equation again:

$$\begin{aligned} & add\ Z\ Z \\ = & \quad \{ \quad \text{applying } add \quad \} \\ & Z \end{aligned}$$

We can now assume that $add\ n\ Z = n$ holds for $n :: Nat$ to show that the inductive step for $S\ n$ also holds:

$$\begin{aligned} & add\ (S\ n)\ Z \\ = & \quad \{ \quad \text{applying } add \quad \} \\ & S\ (add\ n\ Z) \\ = & \quad \{ \quad \text{induction hypothesis} \quad \} \\ & S\ n \end{aligned}$$

This concludes the proof for the second property.

Associativity Let's consider a third property of addition: associativity. We can express the associativity property of *add* as:

$$\forall x\ y\ z :: Nat\ .\quad add\ x\ (add\ y\ z) = add\ (add\ x\ y)\ z$$

We will approach the proof for associativity by induction on x because we cannot reduce any part of the equality as it stands. As usual with induction on natural numbers, we start with the base case where $x = Z$. With proofs for equality, we can start from either side of the equation. We arbitrarily choose to start with the left-hand side as with the previous two proofs:

$$\begin{aligned}
& \mathit{add}\ Z\ (\mathit{add}\ y\ z) \\
= \quad & \{ \quad \text{applying } \mathit{add} \quad \} \\
& \mathit{add}\ y\ z
\end{aligned}$$

Now we are a little stuck as there is no obvious way for us to progress to our goal of $\mathit{add}\ (\mathit{add}\ Z\ y)\ z$. In situations like this, it helps to look at the other side of the equation and begin from there:

$$\begin{aligned}
& \mathit{add}\ (\mathit{add}\ Z\ y)\ z \\
= \quad & \{ \quad \text{applying } \mathit{add} \quad \} \\
& \mathit{add}\ y\ z
\end{aligned}$$

We have now ended up with $\mathit{add}\ y\ z$ by reducing both sides of the equation. We can therefore conclude that the equality holds for $x = Z$ since we showed that both sides can be reduced to the same expression. We could also write this proof as a single series of transformations from one side of the equation to the other:

$$\begin{aligned}
& \mathit{add}\ Z\ (\mathit{add}\ y\ z) \\
= \quad & \{ \quad \text{applying } \mathit{add} \quad \} \\
& \mathit{add}\ y\ z \\
= \quad & \{ \quad \text{unapplying } \mathit{add} \quad \} \\
& \mathit{add}\ (\mathit{add}\ Z\ y)\ z
\end{aligned}$$

Here, "unapplying" means the inverse of "applying": taking the right-hand side of a function definition and replacing it with the corresponding left-hand side. We can now assume that $\mathit{add}\ x\ (\mathit{add}\ y\ z) = \mathit{add}\ (\mathit{add}\ x\ y)\ z$ holds for $x :: \mathit{Nat}$, $y :: \mathit{Nat}$, and

$z :: Nat$ to prove the inductive step for $S\ x$:

$$\begin{aligned}
& add\ (S\ x)\ (add\ y\ z) \\
= & \quad \{ \quad \text{applying } add \quad \} \\
& S\ (add\ x\ (add\ y\ z)) \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
& S\ (add\ (add\ x\ y)\ z) \\
= & \quad \{ \quad \text{unapplying } add \quad \} \\
& (add\ (S\ add\ x\ y)\ z) \\
= & \quad \{ \quad \text{unapplying } add \quad \} \\
& (add\ (add\ (S\ x)\ y)\ z)
\end{aligned}$$

This concludes the proof for associativity.

3.1.2 Understanding structural induction

While you may already be familiar with induction on natural numbers, you may not know that you can perform induction on values of any set which can be defined *inductively*. For example, we could define $\mathbb{N}$ inductively as follows:

- $Z \in \mathbb{N}$
- $\forall n \in \mathbb{N}. S\ n \in \mathbb{N}$

The case for Z is a base case because it does not rely on any other elements of $\mathbb{N}$ for its definition. The case for $S\ n$ is a recursive case, because it is defined in terms of some other element of $\mathbb{N}$ called n. This inductive definition is equivalent to our data type declaration of natural numbers in Haskell:

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

Induction is always performed on a universally-quantified variable in a theorem by breaking the theorem down into several smaller theorems, one for each case that defines the set of values over which the variable ranges.

3.1.3 Induction on lists

As shown above, natural numbers can be defined inductively in a way which closely resembles the corresponding algebraic data type in Haskell.

Lists in Haskell are also defined as an algebraic data type and they can also be defined inductively as:

- $[\,] :: [a]$
- $\forall x :: a . \forall xs :: [a] . x : xs :: [a]$

An algebraic data type for lists could be expressed in Haskell as:

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Empty} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

Note that Haskell's built-in list type is just an algebraic data type as well which is defined exactly like the *List* type above, except with different names for the type and two constructors.

Map fusion In Haskell, it is common to define functions as the composition of several, smaller functions. However, if executed literally, such functions are obviously very inefficient because each function would generate intermediate results which are then immediately consumed by the next function in the chain. The Haskell compiler performs an optimisation known as *fusion* which turns multiple loops over *e.g.* a list into one and eliminates intermediate lists. For example, mapping a function g over a list and then mapping a function f over the resulting list is the same as composing f and g and mapping the resulting function over the list:

$$\forall f :: b \rightarrow c, g :: a \rightarrow b, xs :: [a] . \quad map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$$

Because functions in Haskell are pure, we can prove that this is a valid way to optimise our programs. Our proof is by induction on xs. There is only one base case, which is for the empty list $[\,]$:

$$\begin{aligned}
& map\ f\ (map\ g\ [\,]) \\
= \quad & \{ \quad \text{applying } map \quad \} \\
& map\ f\ [\,] \\
= \quad & \{ \quad \text{applying } map \quad \} \\
& [\,] \\
= \quad & \{ \quad \text{unapplying } map \quad \} \\
& map\ (f \circ g)\ [\,]
\end{aligned}$$

Just as with natural numbers, we can perform a proof just by rewriting either side of the equation until both sides are the same expression.

We can now assume that $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$ holds (our induction hypothesis) to prove the inductive step for $x : xs$:

$$
\begin{aligned}
& \mathit{map}\ f\ (\mathit{map}\ g\ (x : xs)) \\
= \quad & \{ \quad \text{applying } \mathit{map} \quad \} \\
& \mathit{map}\ f\ (g\ x : \mathit{map}\ g\ xs) \\
= \quad & \{ \quad \text{applying } \mathit{map} \quad \} \\
& f\ (g\ x) : \mathit{map}\ f\ (\mathit{map}\ g\ xs) \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& f\ (g\ x) : \mathit{map}\ (f \circ g)\ xs \\
= \quad & \{ \quad \text{unapplying } \circ \quad \} \\
& (f \circ g)\ x : \mathit{map}\ (f \circ g)\ xs \\
= \quad & \{ \quad \text{unapplying } \mathit{map} \quad \} \\
& \mathit{map}\ (f \circ g)\ (x : xs)
\end{aligned}
$$

This concludes the proof.

3.2 Proving a simple compiler correct

For our next example, we are going to take what we have learnt about equational reasoning so far in order to show that a simple compiler is *correct*. Our compiler is going to compile expressions in an expression language to programs for a simple stack-based machine. We begin by implementing the types and functions required to represent the two different languages and to compile from one to the other.

3.2.1 The expression language

Let us begin by defining a data type named `Expr` in Haskell to represent expressions of the expression language (the compiler's *source language*):

```
data Expr = Val Int
          | Plus Expr Expr
```

With this data type, we can represent values such as `Plus (Val 4) (Val 8)` and `Plus (Plus (Val 4) (Val 15)) (Val 8)` in Haskell. However, these values of type `Expr` have no meaning. Even though we may have an intuition for what these values represent due to the names that we have given to `Expr`'s data constructors, the names are completely arbitrary and we could have called them anything. To give meaning to values of this type, we need to implement a *denotational semantics*, also known as an *interpreter*:

```haskell
eval :: Expr -> Int
eval (Val n)    = n
eval (Plus l r) = eval l + eval r
```

The `eval` function maps values of type `Expr` to values of type `Int` that represent the meaning we would intuitively associate with values of type `Expr`.

3.2.2 The instruction set

The *target language* of our compiler is programs for a stack-based machine whose instructions are represented by the following data type in Haskell:

```haskell
data Instr = PUSH Int | ADD
```

We also define a couple of type aliases to say that a list of `Instr` values represents a `Program` and that the machine's `Stack` is represented as a list of `Int` values:

```haskell
type Program = [Instr]
type Stack   = [Int]
```

Just like with our source language, values of type `Instr` or `Program` are meaningless until we give them a semantics, for example in the form of a Haskell function that implements an interpreter:

```haskell
exec :: Program -> Stack -> Stack
exec []                   s   = s
exec (PUSH n : p)         s   = exec p (n:s)
exec (ADD    : p) (y : x : s) = exec p (x+y:s)
```

We can see from this definition that the semantics of our programs are as follows. If we have an empty program, we just return the stack we are given. If the next instruction in the program is a `PUSH` instruction, we add its value to the top of the stack and continue executing the rest of the program. If the next instruction in the program is an `ADD` instruction, we take two values `y` and `x` off the top of the stack, add them up, and add the result to the stack we use to execute the rest of the program with.

3.2.3 The compiler

Now that we have data types to represent both expressions in our source language and programs in our target language, we can write a compiler to translate from the source language to the target language:

```
comp :: Expr -> Program
comp (Val n)    = [PUSH n]
comp (Plus l r) = comp l ++ comp r ++ [ADD]
```

Values from our source language are just compiled to `PUSH` instructions. For `Plus` expressions, we compile the left sub-expression, then the right sub-expression, and then add an `ADD` instruction.

3.2.4 Compiler correctness

Now that we have defined types to represent our source and target languages, interpreters for both, as well as a compiler, we can prove that the compiler is correct.

Compiler correctness We say that our compiler is correct if evaluating an expression e has the same result as executing the program that results from compiling an expression e:

$$\forall e :: Expr, s :: Stack \;.\quad eval\ e : s = exec\ (comp\ e)\ s$$

Since there are infinitely-many possible values of `Expr`, the proof is by structural induction on e. The base case is for values $Val\ n$. That is we want to show that:

$$\forall s :: Stack, n :: Int \;.\quad eval\ (Val\ n) : s = exec\ (comp\ (Val\ n))\ s$$

As usual, we pick one side of the equation and start to simplify it:

$$\begin{aligned} & eval\ (Val\ n) : s \\ = & \quad \{ \text{ applying } eval \ \} \\ & n : s \end{aligned}$$

As we are kind of stuck at this point with just $n : s$, we decide to pick up with the other side of the equation and simplify that:

$$\begin{aligned} & exec\ (comp\ (Val\ n))\ s \\ = & \quad \{ \text{ applying } comp \ \} \\ & exec\ [PUSH\ n]\ s \\ = & \quad \{ \text{ applying } exec \ \} \\ & exec\ []\ (n : s) \\ = & \quad \{ \text{ applying } exec \ \} \\ & n : s \end{aligned}$$

As shown, both sides of the equation can be simplified to the same expression. Therefore the base case holds. The inductive step of our proof is to show that the following holds:

$$\forall s :: \mathit{Stack}\ .\quad \mathit{eval}\ (\mathit{Plus}\ l\ r) : s = \mathit{exec}\ (\mathit{comp}\ (\mathit{Plus}\ l\ r))\ s$$

Since the *Plus* constructor has two parameters of type *Expr*, we have two induction hypotheses, one for *l* and one for *r*:

$$\begin{aligned} \textbf{IH1} :&\quad \forall s :: \mathit{Stack}\ .\quad \mathit{eval}\ l : s = \mathit{exec}\ (\mathit{comp}\ l)\ s \\ \textbf{IH2} :&\quad \forall s :: \mathit{Stack}\ .\quad \mathit{eval}\ r : s = \mathit{exec}\ (\mathit{comp}\ r)\ s \end{aligned}$$

We can now prove that the inductive step is also true by simplifying both sides of the equation until we end up with the same expression. Let's start with the left side:

$$\begin{aligned} & \mathit{eval}\ (\mathit{Plus}\ l\ r) : s \\ = & \qquad \{ \quad \text{applying } \mathit{eval} \quad \} \\ & (\mathit{eval}\ l + \mathit{eval}\ r) : s \end{aligned}$$

We are quickly stuck here with no obvious way to proceed. Let's try the right side of the equation instead:

$$\begin{aligned} & \mathit{exec}\ (\mathit{comp}\ (\mathit{Plus}\ l\ r))\ s \\ = & \qquad \{ \quad \text{applying } \mathit{comp} \quad \} \\ & \mathit{exec}\ (\mathit{comp}\ l +\!\!+\ \mathit{comp}\ r +\!\!+ [\mathit{ADD}])\ s \end{aligned}$$

Again, we get stuck fairly quickly here. Unfortunately, as in previous examples, we have not ended up with the same expression by simplifying both sides of the equation. However, this is an inductive proof and we are in the inductive case, so we have the induction hypotheses available – maybe they could help us? Unfortunately, this is also not the case. If we look at **IH1** and **IH2**, we can see that neither is applicable to the expression we have right now. We need to find a way to transform our expression so that the induction hypotheses can be applied to it.

Distributivity lemma Often when proving more complicated properties about functions, we depend on other properties of functions to help us out. One such property which states that executing a program where a list of instructions *xs* is followed by a list of instructions *ys* is the same as first executing *xs* and then executing *ys* with the stack that results from executing *xs*:

$$\forall xs\ ys :: \mathit{Program}, s :: \mathit{Stack}\ .\quad \mathit{exec}\ (xs +\!\!+ ys)\ s = \mathit{exec}\ ys\ (\mathit{exec}\ xs\ s)$$

The proof for this lemma is by induction on *xs*. As usual with induction on lists, the base case is for the empty list [ ]:

$$\forall ys :: \mathit{Program}, s :: \mathit{Stack}\ .\quad \mathit{exec}\ ([\,] +\!\!+ ys)\ s = \mathit{exec}\ ys\ (\mathit{exec}\ [\,]\ s)$$

Let us start with the right side of the equation where there is only one step to perform before we cannot simplify the expression any further:

$$
\begin{aligned}
& \textit{exec ys } (\textit{exec } [\,]\ s) \\
= \quad & \{ \quad \text{applying inner } \textit{exec} \quad \} \\
& \textit{exec ys s}
\end{aligned}
$$

The left side of the equation is similarly easy and follows directly from the definition of $+\!\!+$:

$$
\begin{aligned}
& (\textit{exec } ([\,] +\!\!+ ys)\ s) \\
= \quad & \{ \quad \text{applying } +\!\!+ \quad \} \\
& \textit{exec ys s}
\end{aligned}
$$

With this, we have proved the base case. We can now move on to the inductive step:

$$\forall ys :: \textit{Program}, s :: \textit{Stack}, x :: \textit{Instr} . \quad \textit{exec } ((x : xs) +\!\!+ ys)\ s = \textit{exec ys } (\textit{exec } (x : xs)\ s)$$

Our induction hypothesis is that the lemma holds for *xs*:

$$\forall ys :: \textit{Program}, s :: \textit{Stack} . \quad \textit{exec } (xs +\!\!+ ys)\ s = \textit{exec ys } (\textit{exec xs s})$$

Let us again being with the right side of the equation:

$$\textit{exec ys } (\textit{exec } (x : xs)\ s)$$

We are immediately stuck here, because we cannot simplify the expression any further without knowing what x is. To deal with this problem, we simply perform case-analysis on x. We begin with the case where $x = \textit{PUSH } n$:

$$
\begin{aligned}
& \textit{exec ys } (\textit{exec } (\textit{PUSH } n : xs)\ s) \\
= \quad & \{ \quad \text{applying inner } \textit{exec} \quad \} \\
& \textit{exec ys } (\textit{exec xs } (n : s)) \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& \textit{exec } (xs +\!\!+ ys)\ (n : s) \\
= \quad & \{ \quad \text{unapplying } \textit{exec} \quad \} \\
& \textit{exec } (\textit{PUSH } n : (xs +\!\!+ ys))\ s \\
= \quad & \{ \quad \text{unapplying } +\!\!+ \quad \} \\
& \textit{exec } ((\textit{PUSH } n : xs) +\!\!+ ys)\ s
\end{aligned}
$$

The inductive step holds for the case where we have a *PUSH* instruction. What about the case where we have an *ADD* instruction:

$$\textit{exec ys } (\textit{exec } (\textit{ADD} : xs)\ s)$$

We immediately have a problem. The third equation of *exec* (which deals with *ADD* instructions) requires there to be at least two elements on the stack. We could perform induction on *s* at this point to explore values for the stack, but we do not need to do that to notice that this will not work. *ADD* requires at least two values on the stack, so if we perform induction we will end up with cases where the stack is empty or only contains one element. In fact, the *exec* function is partial and not every input is mapped to a result as is the case here. Is our compiler wrong? Well, the problem is solely in the *exec* function. Our compiler *comp*, however, will never actually generate code where not at least two values are pushed onto the stack before an *ADD* instruction. We could prove that this is the case, but it would be rather complicated. So instead, we will just continue with the assumption that the stack has at least two elements:

$$
\begin{aligned}
& \mathit{exec}\ \mathit{ys}\ (\mathit{exec}\ (\mathit{ADD} : \mathit{xs})\ s) \\
= \quad & \{ \quad \text{assume } s = b : a : s' \quad \} \\
& \mathit{exec}\ \mathit{ys}\ (\mathit{exec}\ (\mathit{ADD} : \mathit{xs})\ (b : a : s') \\
= \quad & \{ \quad \text{applying } \mathit{exec} \quad \} \\
& \mathit{exec}\ \mathit{ys}\ (\mathit{exec}\ \mathit{xs}\ (a + b : s') \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& \mathit{exec}\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys})\ (a + b : s') \\
= \quad & \{ \quad \text{unapplying } \mathit{exec} \quad \} \\
& \mathit{exec}\ (\mathit{ADD} : (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}))\ (b : a : s') \\
= \quad & \{ \quad \text{unapplying } \mathbin{+\!\!+} \quad \} \\
& \mathit{exec}\ ((\mathit{ADD} : \mathit{xs}) \mathbin{+\!\!+} \mathit{ys})\ (b : a : s') \\
= \quad & \{ \quad \text{assumption for } s \quad \} \\
& \mathit{exec}\ ((\mathit{ADD} : \mathit{xs}) \mathbin{+\!\!+} \mathit{ys})\ s
\end{aligned}
$$

This concludes the proof for the inductive case where we have an *ADD* instruction and it also concludes the proof for our distributivity lemma. We can now return to our main proof where we were stuck on:

$$\mathit{exec}\ (\mathit{comp}\ l \mathbin{+\!\!+} \mathit{comp}\ r \mathbin{+\!\!+} [\mathit{ADD}])\ s$$

With the help of our distributivity lemma, we can now rewrite this expression into a suitable form for the induction hypotheses:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{comp}\ l \mathbin{+\!\!+} \mathit{comp}\ r \mathbin{+\!\!+} [\mathit{ADD}])\ s \\
= & \quad \{ \quad \text{associativity of } \mathbin{+\!\!+} \quad \} \\
& \mathit{exec}\ (\mathit{comp}\ l \mathbin{+\!\!+} (\mathit{comp}\ r \mathbin{+\!\!+} [\mathit{ADD}]))\ s \\
= & \quad \{ \quad \text{distributivity lemma} \quad \} \\
& \mathit{exec}\ (\mathit{comp}\ r \mathbin{+\!\!+} [\mathit{ADD}])\ (\mathit{exec}\ (\mathit{comp}\ l)\ s) \\
= & \quad \{ \quad \text{distributivity lemma} \quad \} \\
& \mathit{exec}\ [\mathit{ADD}]\ (\mathit{exec}\ (\mathit{comp}\ r)\ (\mathit{exec}\ (\mathit{comp}\ l)\ s)) \\
= & \quad \{ \quad \text{induction hypothesis } \textbf{IH1} \quad \} \\
& \mathit{exec}\ [\mathit{ADD}]\ (\mathit{exec}\ (\mathit{comp}\ r)\ (\mathit{eval}\ l : s)) \\
= & \quad \{ \quad \text{induction hypothesis } \textbf{IH2} \quad \} \\
& \mathit{exec}\ [\mathit{ADD}]\ (\mathit{eval}\ r : (\mathit{eval}\ l : s)) \\
= & \quad \{ \quad \text{applying } \mathit{exec} \quad \} \\
& \mathit{exec}\ [\,]\ ((\mathit{eval}\ l + \mathit{eval}\ r) : s) \\
= & \quad \{ \quad \text{applying } \mathit{exec} \quad \} \\
& (\mathit{eval}\ l + \mathit{eval}\ r) : s
\end{aligned}
$$

This expression now exactly matches the one we obtained by rewriting the left side of the equation earlier. Therefore, the inductive step is proved and the proof is complete.

3.3 Calculating a faster compiler

We have shown that our compiler `comp` is correct, but our current implementation is very slow. To illustrate this, let us use reduction steps as a metric for performance. For example, if compile the expression `Plus (Plus (Val 4) (Val 15)) (Val 8)` using `comp`, it would require 15 reduction steps:

```
   comp (Plus (Plus (Val 4) (Val 15)) (Val 8))
=> comp (Plus (Val 4) (Val 15)) ++ (comp (Val 8) ++ [ADD])
=> (comp (Val 4) ++ (comp (Val 15) ++ [ADD])) ++
   (comp (Val 8) ++ [ADD])
=> ([PUSH 4] ++ (comp (Val 15) ++ [ADD])) ++
   (comp (Val 8) ++ [ADD])
=> (PUSH 4 : ([] ++ (comp (Val 15) ++ [ADD]))) ++
   (comp (Val 8) ++ [ADD])
=> (PUSH 4 : (comp (Val 15) ++ [ADD])) ++
   (comp (Val 8) ++ [ADD])
=> (PUSH 4 : ([PUSH 15] ++ [ADD])) ++
   (comp (Val 8) ++ [ADD])
=> (PUSH 4 : (PUSH 15 : ([] ++ [ADD]))) ++
   (comp (Val 8) ++ [ADD])
=> (PUSH 4 : (PUSH 15 : [ADD])) ++
   (comp (Val 8) ++ [ADD])
== (PUSH 4 : (PUSH 15 : (ADD : []))) ++
   (comp (Val 8) ++ [ADD])
=> PUSH 4 : ((PUSH 15 : (ADD : [])) ++
   (comp (Val 8) ++ [ADD]))
=> PUSH 4 : (PUSH 15 : ((ADD : []) ++
   (comp (Val 8) ++ [ADD])))
=> PUSH 4 : (PUSH 15 : (ADD : ([] ++
   (comp (Val 8) ++ [ADD]))))
=> PUSH 4 : (PUSH 15 : (ADD :
   (comp (Val 8) ++ [ADD])))
=> PUSH 4 : (PUSH 15 : (ADD :
   ([PUSH 8] ++ [ADD])))
=> PUSH 4 : (PUSH 15 : (ADD :
   (PUSH 8 : ([] ++ [ADD]))))
=> PUSH 4 : (PUSH 15 : (ADD :
   (PUSH 8 : [ADD])))
== [PUSH 4, PUSH 15, ADD, PUSH 8, ADD]
```

As we can see, a lot of reduction steps are spent evaluating the results of applying `(++)`. This is because `(++)` requires a number of evaluation steps linear to the size of the first argument. For example, for the empty list, it requires one evaluation step:

```
   [] ++ ys
=> ys
```

For a list with one element, it requires two evaluation steps:

```
   [x] ++ ys
=> x : ([] ++ ys)
=> x : ys
```

Indeed, the number of evaluation steps required to reduce `xs ++ ys` for all lists `xs` and `ys` is the number of elements in the first argument + 1, i.e. `length xs + 1`.

So far we have seen how to use induction to prove properties (or equalities) about our programs. However, we can also use induction *constructively*. To illustrate this, we are going to use induction to calculate a faster version of `comp` which requires fewer reduction steps for every given input. This process is known as *constructive induction*. The first step of this process is to come up with a *specification* that should be satisfied by the function we are trying to construct. For our new, faster compiler, we wish to construct a new function `comp'` so that the following specification holds:

$$\forall e :: Expr, p :: Program \; . \quad comp' \; e \; p = comp \; e +\!\!+ \; p$$

Coming up with such a specification is the difficult part of the constructive induction process. For our `comp` function, we observed that our use of the $+\!\!+$ operator contributes significantly to the number of reduction steps required to compile an expression. Therefore, our specification for `comp'` effectively says that it should combine the behaviour of `comp` and $+\!\!+$. Our hope here is that the definition of `comp'` will be able to construct the resulting programs more efficiently. Let's see what happens.

Once we have a specification, the next step is come up with the definition for the function we are trying to construct. In this case, we will do that by induction on e, leaving us with the following two cases:

$$\begin{array}{lll} \forall p :: Program \; . \quad comp' \; (Val \; n) \; p & = & comp \; (Val \; n) +\!\!+ \; p \\ \forall p :: Program \; . \quad comp' \; (Plus \; l \; r) \; p & = & comp \; (Plus \; l \; r) +\!\!+ \; p \end{array}$$

The left-hand sides of the two equations will form the left-hand sides of the definition of `comp'`:

```haskell
comp' :: Expr -> Program -> Program
comp' (Val n)      p = ???
comp' (Plus l r) p = ???
```

All we need to do now is figure out what the right-hand sides of the two equations should be. We do this by taking the right-hand sides of the two cases shown above and reducing them as much as possible. For example, if we take the right-hand side of *Val n* case we can simplify it as follows:

$$\begin{array}{ll} & comp \; (Val \; n) +\!\!+ \; p \\ = & \{ \quad \text{applying } comp \quad \} \\ & [PUSH \; n] +\!\!+ \; p \\ = & \{ \quad \text{applying } +\!\!+ \quad \} \\ & PUSH \; n : ([\,] +\!\!+ \; p) \\ = & \{ \quad \text{applying } +\!\!+ \quad \} \\ & PUSH \; n : p \end{array}$$

The expression we have ended up with cannot be reduced any further and, therefore, we will now use it as the right-hand side of the first equation that defines `comp'`:

```haskell
comp' :: Expr -> Program -> Program
comp' (Val n)    p = PUSH n : p
comp' (Plus l r) p = ???
```

For the second equation, we start with *comp* $(Plus\ l\ r) +\!\!+\ p$ and follow the same process. However, it is essential to note that we are performing induction and this is an inductive case. Therefore, we have two inductive hypotheses:

$$\begin{array}{lllll} \textbf{IH1} & \forall p :: Program\ . & comp'\ l\ p & = & comp\ l +\!\!+ p \\ \textbf{IH2} & \forall p :: Program\ . & comp'\ r\ p & = & comp\ r +\!\!+ p \end{array}$$

We can now begin simplifying the expression:

$$\begin{array}{ll} & comp\ (Plus\ l\ r) +\!\!+ p \\ = & \{\ \text{applying } comp\ \} \\ & (comp\ l +\!\!+ (comp\ r +\!\!+ [ADD])) +\!\!+ p \end{array}$$

At this point, we could apply the inductive hypotheses as follows:

$$\begin{array}{ll} & (comp\ l +\!\!+ (comp\ r +\!\!+ [ADD])) +\!\!+ p \\ = & \{\ \text{applying } \textbf{IH2}\ \} \\ & (comp\ l +\!\!+ (comp'\ r\ [ADD])) +\!\!+ p \\ = & \{\ \text{applying } \textbf{IH1}\ \} \\ & (comp'\ l\ (comp'\ r\ [ADD])) +\!\!+ p \end{array}$$

However, the expression we end up with by taking this approach still contains $+\!\!+$ and cannot be simplified any further. We can do better. In Exercise 95, you prove the associativity of the $+\!\!+$ operator. With the help of that property, we can rewrite the initial expression slightly so that we can eliminate the occurrence of the $+\!\!+$ operator before applying the two inductive hypotheses:

$$\begin{array}{ll} & (comp\ l +\!\!+ (comp\ r +\!\!+ [ADD])) +\!\!+ p \\ = & \{\ \text{associativity of } +\!\!+\ \} \\ & comp\ l +\!\!+ ((comp\ r +\!\!+ [ADD]) +\!\!+ p) \\ = & \{\ \text{associativity of } +\!\!+\ \} \\ & comp\ l +\!\!+ (comp\ r +\!\!+ ([ADD] +\!\!+ p)) \\ = & \{\ \text{applying } +\!\!+\ \} \\ & comp\ l +\!\!+ (comp\ r +\!\!+ (ADD : ([\,] +\!\!+ p))) \\ = & \{\ \text{applying } +\!\!+\ \} \\ & comp\ l +\!\!+ (comp\ r +\!\!+ (ADD : p)) \\ = & \{\ \text{applying } \textbf{IH2}\ \} \end{array}$$

$$\begin{aligned}
& \quad comp\ l +\!\!+ (comp'\ r\ (ADD : p)) \\
= & \quad \{ \quad \text{applying } \textbf{IH1} \quad \} \\
& \quad comp'\ l\ (comp'\ r\ (ADD : p))
\end{aligned}$$

The expression we end up with here does no longer contain the $+\!\!+$ operator and we can now happily use it in our definition of `comp'`:

```haskell
comp' :: Expr -> Program -> Program
comp' (Val n)     p = PUSH n : p
comp' (Plus l r) p = comp' l (comp' r (ADD : p))
```

To test that this is actually more efficient than our definition of `comp`, let us try to compile the same expression we compiler earlier using `comp` but this time using `comp'` where we simply provide the empty program `[]` as the second argument:

```
   comp' (Plus (Plus (Val 4) (Val 15)) (Val 8)) []
=> comp' (Plus (Val 4) (Val 15)) (comp' (Val 8) (ADD : []))
=> comp' (Plus (Val 4) (Val 15)) (PUSH 8 : (ADD : []))
=> comp' (Val 4) (comp' (Val 15) (PUSH 8 : (ADD : [])))
=> comp' (Val 4) (PUSH 15 : (PUSH 8 : (ADD : [])))
=> PUSH 4 : (PUSH 15 : (PUSH 8 : (ADD : [])))
== [PUSH 4, PUSH 15, ADD, PUSH 8, ADD]
```

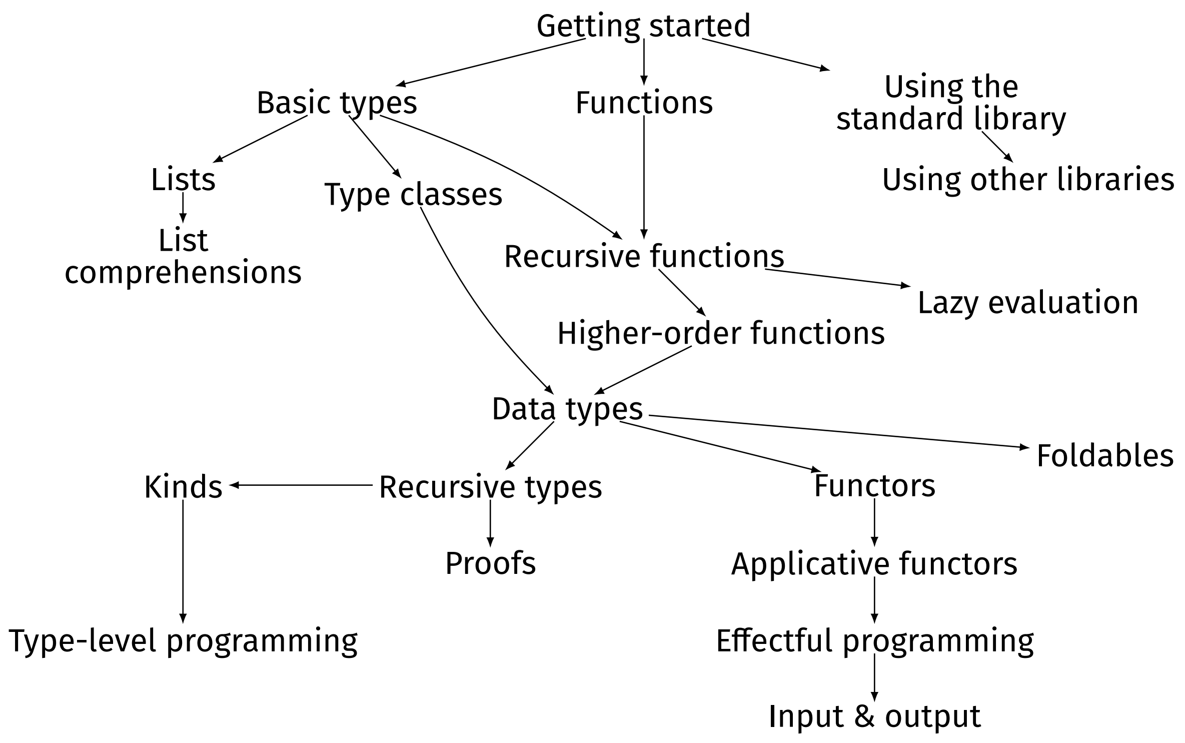
As we can see, the program can now be compiled in 5 reduction steps as opposed to 15 reduction steps that were required previously with `comp`. Furthermore, we can now redefine `comp` in terms of `comp'`:

```haskell
comp :: Expr -> Program
comp e = comp' e []
```

λ.4

Exercises

This chapter contains exercises for the module. Each section in this chapter corresponds to one topic from the lectures. You can find recommendations for which sets of exercises to complete after a given lecture on the module website and recommendations for each week of term in Section 1.2. We strongly recommend that you try to keep up with the exercises as they are very valuable for the coursework. If you are not following the order in which the exercises are presented in this guide, the graph below shows prerequisite dependencies between topics in the module:



CS141 lab sessions should primarily be fun! Make use of them to practice the skills we cover in the lectures and to ask us questions. We do take a register to keep track of attendance, but if you should miss a lab *e.g.* due to illness, that's not an issue and you are welcome to attend an online lab at a later time instead.

4.1 Getting started

The purpose of these first exercises is twofold: firstly, you will set up the tools that we use as part of the module and become acquainted with the basics of how they work; secondly you will encounter, compile, and improve an existing Haskell program. For these exercises, there is no expectation that you understand all of the code you see in front of you or that you accomplish anything in particular by the end of it. Indeed, these exercises are suitable for you even if you have not been to any of the lectures yet. Looking at existing codebases and changing a few values here and there is a great way to get a feel for what programs written in a particular programming language look like and, in the case of this module, what to expect later on.

Ex1 If you have not done so yet, you should set up your Haskell development environment now (see Section 2.1 and Section 2.2). Most of the tools you need are already pre-installed on the departmental machines, but you must open a shell (with the Terminal application on the lab machines) and run the following command to make the `stack` tool available on your user account (this only needs to be done once):

```
$ /modules/cs141/haskell-setup.sh
```

Ex2 There is some skeleton code for most of the lab sessions, including this one. You can obtain the code for this practical by cloning it from GitHub which you can do by running the following command in your terminal window:

```
$ git clone https://github.com/fpclass/lab-getting-started
```

By default, this will create a folder named `lab-getting-started` in the current working directory (your home directory, by default) with the skeleton code in it. Once you have cloned the repository, you may wish to verify that it compiles without any problems:

```
$ cd lab-getting-started
$ stack run
```

If everything goes well, you should see some output along the lines of:

```
Building all executables for 'hatch' once.
hatch-0.2.0.0: configure (exe)
Configuring hatch-0.2.0.0...
hatch-0.2.0.0: build (exe)
Preprocessing executable 'hatch' for hatch-0.2.0.0..
Building executable 'hatch' for hatch-0.2.0.0..
[1 of 7] Compiling Layout
[2 of 7] Compiling Paths_hatch
[3 of 7] Compiling Transforms
[4 of 7] Compiling Images
[5 of 7] Compiling Hatch
[6 of 7] Compiling Lab
[7 of 7] Compiling Main
Linking .stack-work/dist/../build/hatch/hatch ...
hatch-0.2.0.0: copy/register
Installing executable hatch in /dcs/../bin
```

A window should open with lovely animations. If so, congratulations: you have compiled and run your first Haskell program! The `stack` tool is correctly installed and works as expected – you are now ready to work on the exercises! If you are not seeing a window with lovely animations, ask one of the lab tutors for assistance.

There is a `src/Lab.hs` file in the `lab-getting-started` directory which contains some definitions responsible for rendering the lovely animation. You should open that file in your text editor of choice (see Chapter 2 for information about the text editors available to you). If you are using VSCode, the `haskell-setup.sh` script you ran earlier will already have installed some Haskell-related plugins.

Ex3 Open the `src/Lab.hs` file in your preferred text editor. Take a look at the definition of `animation`. At this point, you are not expected to understand everything you see and the purpose of this exercise is simply to experiment by changing code that is already there. Not everything you try may work, but that's okay! Here are some examples of things you could try before moving on to the next exercise sheet:

- Turn the spinning cat into a spinning dog.
- Make the goose moonwalk away from the ducks.
- Make the spinning cat spin the other way.
- Make the goose run twice as fast.
- Hide the dog behind a giant duck.

There are a number of pictures available, namely `cat`, `dog`, `duck`, `goose`, and `blank`. The standard mathematical operators (`+`, `-`, `*`) are also available, so you can perform calculations on values.

A number of useful operators and expressions related to positioning and transforming images are also available:

Function / Operator	Description
`img1 <|> img2`	Puts `img2` to the right of `img1`.
`img1 <-> img2`	Puts `img1` above `img2`.
`img1 <@> img2`	Puts `img1` in front of `img2`.
`scale sf img2`	Scales `img` by scale factor `sf`.
`offset tx ty img`	Moves `img`, `tx` pixels right and `ty` pixels up.
`rotate deg img`	Rotates `img` by `deg` degrees.
`mirror img`	Reflects `img` through the y-axis.

Note that the right hand side of `animation` depends on a parameter, `t`, which is the number of frames that have been rendered so far (30 per second typically). Using `t` in our calculations for positions, rotations, etc. allows us to change them as time progresses.

4.2 Functions

Topics: Definitions, basic arithmetic expressions, string values, boolean values, functions, and basic pattern matching.

This is the first "real" set of exercises for the module which cover the content of the second lecture on definitions and functions in Haskell. If you have missed the lecture or want to read over the relevant chapters in the textbooks first, we always point out the recommended reading at the start of an exercise sheet:

📘 *Recommended reading*: Chapters 1 and 2 of *Learn you a Haskell* (Lipovača, 2011) or Chapters 1 and 2 of *Programming in Haskell* (Hutton, 2016).

Remember that *Learn you a Haskell* is freely available online and so it is a great reference of notes for Haskell programming. As with the previous exercises, there is some skeleton code which you can obtain by cloning it from GitHub:

```
$ git clone https://github.com/fpclass/lab-functions
```

Unlike in the previous set of exercises, where we used `stack run` to compile and run the program, we will not actually be creating runnable programs for most of the exercises. Instead you may wish to just compile the code with `stack build`:

```
$ cd lab-functions
$ stack build
```

This will compile the code and any errors that arise will be reported to you so that you can fix them. A very useful tool which you may find helpful for completing the exercises and debugging your code is a Read-Eval-Print Loop (or REPL for short). This is offered by many modern programming languages and development tools, including the Glasgow Haskell Compiler (which `stack` is using behind the scenes). You can launch the REPL by invoking the following command in *e.g.* the `lab-functions` folder:

```
$ stack repl
```

This loads the code for these exercises and allows you to enter arbitrary expressions which the REPL will evaluate for you. You should see something similar to the following prompt:

```
*Lab Lab>
```

Simply enter an expression like `1+1` and hit enter to evaluate it. The REPL will print the result of evaluating the expression:

```
*Lab Lab> 1+1
2
```

There is a `src/Lab.hs` file in the `lab-functions` directory which contains some definitions for this lab. Because we have run `stack repl` in the directory with the code (`lab-functions`), the REPL has automatically loaded the the `src/Lab.hs` file for us, so you can refer to definitions in it:

```
*Lab Lab> name
"Michael"
*Lab Lab> age * 2
58
```

Ex4 At this point you may wish to read Chapter 2 if you have not done so already and set up your text editor to suit your preferences. If you are using VSCode, the `haskell-setup.sh` script you ran earlier will already have installed some Haskell-related plugins for you.

Ex5 Open the `src/Lab.hs` file in your preferred text editor and modify the definitions of `age` and `name` to match your name and age.

Instead of typing an expression which should be evaluated by the REPL, you may also type in a command (all commands start with a colon). The following commands are supported (among others):

`:q`	Quits the REPL.
`:r`	Reloads all files that are currently loaded.
`:l FILENAME`	Loads `FILENAME` into the REPL.

Assuming you did not close the REPL to edit `src/Lab.hs`, it will still be running. The REPL does not automatically check for updates to any files that are currently loaded, so you will have to reload it with the `:r` command. In general, the `:r` command reloads all files that are loaded in the REPL. Now try evaluating `age` and `name` again. They should match whatever values you changed them to.

Ex6 Complete the definitions of `triple`, `tripleV2`, `not`, `and`, `max`, and `perimeterRect` in `src/Lab.hs`. You can test them in the REPL to see if you have got them right. Remember to reload the file once you have made some changes.

Ex7 Some of the above functions, such as `not`, `and`, and `max`, can be defined in many different ways. Aside from your current definitions for them, can you think of one additional way to define each?

Ex8 The skeleton code for these exercises (and for many of the other exercise sheets) also comes with a test suite which you can use to test the correctness of your functions.

Simply run `stack test` in a terminal window to run all unit tests against your code. You should make sure that all tests succeed. You should see somewhere close to the end of the output:

```
7 examples, 0 failures
```

If any tests fail, there will be an explanation of why they failed.

Ex9 On paper (or equivalent), trace the evaluation of

```haskell
min (perimeterRect 4 8) (perimeterRect 10 2)
```

Ex10 With a friend, another student of your choice, or within a small group: each pick one topic from the lectures so far that you found confusing, then get one of the others to try and explain it after a few minutes of preparation.

Ex11 With a friend, another student of your choice, or within a small group: discuss whether...

- ...you think reduction or mutation is easier to understand?
- ...you prefer to use indentation (Haskell, Python, ...) or curly brackets (Java, C, ...) to denote scope and why?

4.3 Basic types

Topics: Basic types, function types, parametric polymorphism, lists, pairs, and type class constraints.

These exercises are about *types* in Haskell. In a nutshell, the type of an expression tells you what sort of value an expression eventually evaluates to. Not all expressions can be typed! If an expression cannot be reduced to a normal form, such as `not 7`, then it does not have a type and the compiler will refuse to compile the program. We use the following notation to say that an expression `expression` has type `type` – this is referred to as a *typing*:

```
expression  ::  type
```

Some examples of typings are:

```
True                         :: Bool
'a'                          :: Char
\x -> x                      :: a -> a
\x -> False                  :: a -> Bool
\x -> \y -> y                :: a -> b -> b
if 5 > 6 then 'a' else 'b'   :: Char
42                           :: Num a => a
4 + 8 * 15 - 16              :: Num a => a
(\x -> x) True               :: Bool
```

As you can see, the size or complexity of a term does not necessarily correspond to that of a type. It only matters what an expression evaluates to. Some expressions have multiple permissible types. For example, the expression `42` could have type `Int` (the type of signed integers where the precision depends on your platform), `Integer` (the type of arbitrary precision integers), `Num a => a` (a polymorphic type `a` constrained to all numeric types by the constraint `Num a`), as well as others.

Recommended reading: Chapter 3 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 3 of *Programming in Haskell* (Hutton, 2016).

Types

If an expression can be typed, the Haskell compiler can *infer* the *most general* type for us. For example, for numbers such as `42`, the `Int` type is permissible, but `Num a => a` is a more general type. "More general" generally means "more polymorphic". There is a command in the REPL which we can use to ask for the type of an expression:

`:t EXPRESSION`	Shows the type of `EXPRESSION`.

Ex12 Launch the REPL with `stack repl` in an arbitrary directory that does *not* already contain any Haskell code and try it for yourself by asking for the types of the following expressions with the `:t` command:

- `'a'`
- `[True, True, False]`
- `[1,2,3,4,5]`
- `[]`
- `\x -> x`
- `1+1`

Ex13 Since the Haskell compiler always infers the most general type for an expression, you may sometimes want to validate whether a less general type is also permissible. You can annotate expressions with typings for that purpose and get the Haskell compiler to validate your annotations. For example, try to ask for the types of the following three expressions:

- `108`
- `(108 :: Int)`
- `(True :: Int)`

As you can see, you get the most general type for `108`. For `(108 :: Int)`, the compiler validates that `Int` is indeed a permissible type for `108`. For the last expression, we get a type error since `Int` is not a permissible type for `True`.

Ex14 Some expressions cannot be reduced to values and therefore do not have a type. Try asking for the types of the following expressions. Each expression will result in the type error which is explained below:

- `not 7`
  As mentioned in the lecture on type classes, the literal `7` has the most general type `Num a => a`. The Haskell compiler also knows that the `not` function has type `Bool -> Bool` so it expects its argument to be of type `Bool`. Because `Bool` is less polymorphic than `a`, the Haskell compiler deduces that `7` should be of type `Bool`. However, this results in a constraint of `Num Bool` which cannot be resolved because there is no instance of `Num` for the `Bool` type.

- `[1,True,3]`
  This results in the same error.

- `['a', False]`
  Lists in Haskell are *homogeneous*. That is, they can only contain elements which have the same type. Neither `'a'` nor `False` have a polymorphic type and both have different types. Therefore, the Haskell compiler will tell you that the two types do not match.

Ex15 The REPL essentially wraps each expression you try to evaluate into a call to the `show :: Show a => a -> String` function. Try to evaluate some expressions which *are* well typed, but whose types do not satisfy the `Show` predicate. This is usually the case for functions. For example:

- `not`
  Even though `not` is well typed, you get a type error if you try to evaluate it in the REPL. That is because the REPL does not know how to display a result that is a function. The error you get will tell you that the `Show` constraint cannot be satisfied for `Bool -> Bool`, the type of `not`.

4.4 Using the standard library

Topics: Prelude, modules and packages, using standard library functions.

Like most programming languages, Haskell has a standard library which contains many useful definitions that virtually every Haskell program makes some use of. Haskell's standard library is called the *Prelude*. You can find the documentation for the Prelude on Hackage:

http://hackage.haskell.org/package/base/docs/Prelude.html

Some of the things you see in the documentation there will not make any sense yet, but we will cover most of it as the module progresses. We have already come across some functions from the Prelude, such as `not`. Everything contained in the Prelude is automatically imported into Haskell modules.

Modules Recall that a module is a collection of definitions and each Haskell source file corresponds to a module. Modules have names such as `Lab`, `Prelude`, or `Data.List`. The name of a module must typically match its filename. `Prelude` is the only module that is imported by default. If you want to import other modules, you can use `import` statements:

```haskell
import Data.List
```

This will make everything defined in `Data.List` available in the module you are currently working on. If you do not want to import everything, but only some definitions, you can specify that too. For example, the following `import` statement only imports the `nub` and `sort` functions from `Data.List`:

```haskell
import Data.List (nub, sort)
```

If, instead of explicitly listing which definitions you want, you want to state which ones you do not want, you can do that too. We have already seen this in the Functions lab:

```haskell
import Prelude hiding (not)
```

Even though `Prelude` is automatically imported, we may sometimes not want everything from it. In the case of the Functions lab, we wanted to define the `not` function ourselves, so we used a `hiding` clause in an `import` statement to state the implementation from `Prelude` should not be imported.

You can find some more examples of what is available to you with respect to importing modules on the Haskell Wiki[1].

[1] https://wiki.haskell.org/Import

Packages Collections of Haskell modules can be used to form *packages*. A `.cabal` file is used to describe and configure a package. All of the skeleton projects for the exercises and courseworks have `.cabal` files so you can have a look at those to see what they contain. Packages have names and version numbers. They can also depend on other packages. By default, only the `base` package is added as a dependency to new packages. The `Prelude` module is part of the `base` package. You can view the documentation for `base` on Hackage:

`http://hackage.haskell.org/package/base/`

Ex16 For this exercise, we will be creating a simple package from scratch with the help of `stack`. In a terminal, run the following:

```
$ stack new my-package simple-library --resolver=lts-16.27
```

The `stack new` command is used to create a new package. We have specified `my-package` as the name of the new package and a folder named `my-package` will have been created in the current directory. The `simple-library` argument specifies the template[2] that `stack` should use for the new package. Finally, `–resolver=lts-16.27` specifies the stack resolver we want to use, which we set to the same one that is installed on the departmental machines – we will explain what this is used for later.

Ex17 Explore the `my-package.cabal` and `src/Lib.hs` files that were created. In the `my-package.cabal` file you will find two lines at the top which specify the name and version of the package:

```
name:                my-package
version:             0.1.0.0
```

Further down, you will find a section that describes the build output (a Haskell library):

```
library
  hs-source-dirs:      src
  exposed-modules:     Lib
  build-depends:       base >= 4.7 && < 5
  default-language:    Haskell2010
```

This section tells us that the source code for the package can be found in the `src` directory, that there is one module named `Lib`, and that we have a dependency on the `base` package. There is a version constraint placed on the dependency on the `base` package which states that we will accept versions of `base` that are greater or

[2] `https://github.com/commercialhaskell/stack-templates`

equal to 4.7 and smaller than 5. The version of `base` that is installed is 4.12 so the constraint can be satisfied.

Ex18 Add a new file named `Util.hs` to the `src` directory and write the following into it:

```
module Util where

double :: Int -> Int
double x = x*2
```

Ex19 Open `my-package.cabal` and add your new `Util` module to the list of modules for the `exposed-modules` field:

```
library
  hs-source-dirs:      src
  exposed-modules:     Lib, Util
  build-depends:       base >= 4.7 && < 5
  default-language:    Haskell2010
```

Ex20 In `src/Lib.hs`, import the `Util` module with a suitable `import` statement and create a new definition for a function `quadruple :: Int -> Int` which uses `double` twice to quadruple its argument. Note that the `src/Lib.hs` file that was generated by `stack new` earlier has an *export list* in the module header:

```
module Lib (someFunc) where
```

In order for `quadruple` to be available to other modules that import `Lib` (which includes the REPL), we must either add `quadruple` to the export list:

```
module Lib (someFunc, quadruple) where
```

Alternatively, you could remove the export list entirely

```
module Lib where
```

Ex21 Run `stack repl` in your package directory and use your `quadruple` and `double` functions to verify that they work as you would expect. If everything works as intended, then congratulations! You have created your first package from scratch with two modules and you have successfully imported a function from one of the modules into the other.

4.5 Lists

Topics: Lists, standard library functions on lists.

These exercises are about lists. As usual, you can obtain the skeleton code for the exercises by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-lists
```

While arrays are a built-in data structure in imperative programming languages that can easily be used through built-in features in the languages, linked lists take the same role in functional programming languages. Linked lists consist of elements and each element consists of a head (the value of the element) and a tail (the next element or the empty list). In Haskell, the empty list is represented by `[]` and elements can be added to it with `:` ("cons"). For example, `108 : []` represents a list with one element `108`. This is quite ugly, so there is some syntactic sugar which lets us write lists as *e.g.* `[1,2,3]` which is equivalent to `1 : 2 : 3 : []` which is equivalent to `1 : (2 : (3 : []))`.

Recommended reading: Chapter 2 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 3.3 of *Programming in Haskell* (Hutton, 2016).

Ex22 There are three definitions for `numbers`, `numbers'`, and `numbers''` which show off a list without explicit brackets, with explicit brackets, and using the syntactic sugar respectively. Add the missing elements to `numbers'` and `numbers''` so that those two definitions represent the same list as `numbers`.

Ex23 With the help of the `reverse` function from the standard library, which reverses lists, complete the definition of `isPalindrome` which checks whether a list of characters is a palindrome. For example:

```
isPalindrome "radar" ==> True
isPalindrome "2020"  ==> False
```

Just like lists are constructed by "cons"-ing elements to smaller lists, starting with the empty list, lists can be deconstructed through pattern-matching by "uncons"-ing elements from the start of a list. For example, the `head` function returns the first element of a list and discards the rest:

```
head :: [a] -> a
head (x:xs) = x
```

We can have nested patterns so that *e.g.* the following function will retrieve the second element of a list:

```
headOfTail :: [a] -> a
headOfTail (x:y:xs) = y
```

Note that this is not a function you should ever write though, since it will crash for any list that does not have at least two elements. The `head` function will also crash if it is given a list that does not have at least one element.

Ex24 Complete the definition of `validModuleCode` which should determine whether a given argument is a valid Computer Science module code. For the purpose of this exercise, a valid module code consists of five characters: the letters `'c'` and `'s'`, followed by three digits. You may find the `isDigit` function from `Data.Char` useful. For example:

```
validModuleCode "cs141" => True
validModuleCode "cs263" => True
validModuleCode "cs407" => True
validModuleCode "es141" => False
validModuleCode "cakes" => False
validModuleCode "lie"   => False
```

4.6 Type classes

Topics: Type classes and type class instances.

These exercises are about type classes in Haskell. You can obtain the skeleton code by cloning the corresponding repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-type-classes
```

Type classes in Haskell are used to constrain parametric polymorphism. This is useful because ordinarily, when we have a type variable, we know nothing about the type that may be instantiated for it. For example, in the definition of the identity function all we can do with `x` is return it:

```haskell
id :: a -> a
id x = x
```

We cannot do anything else with `x` since we know nothing about it. For example, if we wanted to define a function which appends some text to the result of `show`ing a value, we could not write:

```haskell
showAndAppend :: a -> String
showAndAppend x = "Value of x is:" ++ show x
```

This is a type error because we do not know that `show` is defined for whatever type `a` gets instantiated with when we use this `showAndAppend` function. Instead, we need to constrain `a` to only those types which implement the `Show` type class (whose interface contains the `show` function):

```haskell
showAndAppend :: Show a => a -> String
showAndAppend x = "Value of x is:" ++ show x
```

This now compiles since `showAndAppend` is now only permitted to be used with arguments of types that are instances of the `Show` type class. Note that, as usual, we do not need to specify the type signature explicitly and the compiler would be able to infer the type `showAndAppend` automatically. In other words, if we just write:

```haskell
showAndAppend x = "Value of x is:" ++ show x
```

This would still compile, since the compiler would infer the correctly constrained type signature for us. For a type to satisfy a type class constraint, it must implement the interface that is described by the type class. Therefore, if we have a value of some type `a` that is constrained by some type class constraint, then we know that whatever type gets instantiated for `a` will at least support all the functions and operations described by the type class. In order to show that a particular type can satisfy a type class constraint, we must implement a type class instance for it which implements the respective interface.

📕 *Recommended reading*: Chapter 3 of *Learn you a Haskell* (Lipovača, 2011) or Chapters 3 and 8.5 of *Programming in Haskell* (Hutton, 2016).

💡 Do not forget to test your solutions with `stack test` as you progress through the exercises.

Functional programming is firmly grounded in mathematics and therefore mathematical abstractions are often useful programming abstractions in Haskell. Abstractions are often represented by type classes. Two examples of this we consider for the following exercises are algebraic structures known as *semigroups* and *monoids*. Type classes for both of these structures exist in Haskell's standard library, but for these exercises we will implement them both ourselves.

Semigroups

A semigroup is any type which has an associative binary operation. We can define a type class for this where the binary operator is called `<>`:

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

The associativity law for `<>` can be described as follows, although note that there is no way to tell Haskell about this law or get the compiler to check that our instances of the `Semigroup` class satisfy it. Later on in the module we will see how to prove by hand that such laws hold:

Associativity $x <> (y <> z) \quad = \quad (x <> y) <> z$

We say that a type *forms* a semigroup if there is an instance of the `Semigroup` type class for it which obeys the associativity law.

Ex25 Implement `(<>)` of the `Semigroup` instance for `Int` so that it obeys the associativity law. Note that there are at least two possible implementations which satisfy the associativity laws – can you think of at least two?

Ex26 Implement `(<>)` of the `Semigroup` instance for `[a]` so that it obeys the associativity law.

Ex27 Once implemented, you can test that your `Semigroup` instances satisfy the associativity law by running `stack test`. You can also play with them in the REPL: for example, to test the `[a]` instance:

```
*Lab Lab> [1,2,3] <> [4,5,6]
[1,2,3,4,5,6]
*Lab Lab> [True, False] <> [True]
[True, False, True]
*Lab Lab> "hello" <> "there"
"hellothere"
```

Monoids

A monoid is an algebraic structure which is a semigroup and additionally has a unit value. In Haskell, we can declare a type class for types which are monoids:

```
class Semigroup a => Monoid a where
  mempty :: a
```

As we can see, we have a super class constraint which says that in order for some type `a` to be a `Monoid`, it must first be an instance of `Semigroup`. Instances of the `Monoid` type class should obey the following *monoid laws*:

Left identity	$mempty <> x \;=\; x$
Right identity	$x <> mempty \;=\; x$

We say that a type *forms* a monoid if there is an instance of the `Monoid` type class for it which obeys the monoid laws.

Ex28 Implement `mempty` of the `Monoid` instance for `Int` so that it obeys the monoid laws. Remember that there are at least two possible implementations for `Semigroup` for `Int` – how do they affect our choice for the implementation of `mempty`?

Ex29 Implement `mempty` of the `Monoid` instance for `[a]` so that it obeys the monoid laws.

The following exercises make use of *higher-order functions*. You may wish to complete the exercises in Section 4.9 first and then return to the remaining exercises here.

Ex30 Functions of type `a -> b` form a monoid if `b` is a monoid. Implement the `Semigroup` and `Monoid` instances for `a -> b` so that they obey the semigroup and monoid laws. Note that there are no unit tests for this task as it would require tests for function equality.

If this seems a bit unintuitive, you can think of this as a way of composing two functions of type `a -> b` to yield a new function of type `a -> b` which takes an input of type `a`, gives it to both original functions which results in two values of type `b` that are then combined into one value of type `b` through the implementation of `(<>)` for `b`. For example:

```
*Lab Lab> ((\x -> x ++ [1,2]) <> (\y -> y ++ [3,4])) [5]
[5,1,2,5,3,4]
```

4.7 List comprehensions

Topics: List comprehensions.

This practical is about list comprehensions in Haskell. You can obtain the code for this lab by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-list-comprehensions
```

📘 *Recommended reading*: Chapter 2 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 5 of *Programming in Haskell* (Hutton, 2016).

Haskell has some syntactic sugar for generating lists which is referred to as *list comprehensions*. These expressions are very similar to set comprehensions in mathematics. You will learn about their syntax and define some lists using them.

In the first lecture, you saw the `[1..4]` notation to generate the list of numbers from 1 to 4: `[1,2,3,4]`. In general, the `[n..m]` syntax can be used to denote ranges between any two values `n` and `m`. Note that for this to work, `n` and `m` must have the same type and that type must satisfy an `Enum` constraint. Specifically, `[n..m]` is syntactic sugar for `enumFromTo n m` (a member of the `Enum` type class, which we will learn about later in the module).

Ex31 Complete the definitions of `zeroToTen` and `fourToEight` in `Lab.hs` to implement the lists representing the numbers from 0 to 10 and from 4 to 8 respectively.

Ex32 As mentioned above, the `[n..m]` syntax works for any type that can satisfy an `Enum` constraint. Complete the definition of `lowercase` in `Lab.hs` to implement the list of lower-case characters from `'a'` to `'z'` without explicitly listing all the characters.

List comprehensions can also be used to generate lists from other lists, just like set comprehensions are used to generate sets from other sets. For example, the following expression is a list comprehension which generates the list of numbers that are double the numbers from 0 to 10:

```
[2*n | n <- [0..10]]
```

The `n <- [0..10]` part in this example is referred to as a *generator*. Given some list on the right of `<-`, it loops through all the elements of that list and binds them to `n` one after the other. The part on the left of the `|` is what is used to generate an element of the resulting list, for all values obtained from the right of the `|`. So, for this example, the resulting list is `[0,2,4,6,8,10,12,14,16,18,20]`.

Ex33 Using a list comprehension, complete the definition of `powersOfTwo`, which calculates the powers of two for the factors from 1 to 10. The exponentiation operator in Haskell is ^.

Ex34 Using a list comprehension, complete the definition of `factorials`, which calculates the list of factorials for the numbers 1 to 10.

List comprehensions may contain more than one generator. If there is more than one generator, all subsequent generators loop through their elements for every element in the preceding generator. You can think of these as nested loops:

```
[x*y | x <- [1..3], y <- [1..4]]
```

In this example, there are three elements in the list used by the first generator. Each element in that list is multiplied with every element in the list used by the second generator. Therefore, the result is a list of 12 elements where the first four elements are multiples of one, the next four are multiples of two, and the last four are multiples of three: `[1,2,3,4,2,4,6,8,3,6,9,12]`.

Ex35 Using a list comprehension with two generators, define `coords` to be a list of coordinates where the top left corner of the coordinate system is `(0,0)` and the bottom right is `(10,10)`.

Ex36 Generators after the first in a list comprehension may also refer to variables that are bound by preceding generators. Use this to complete the definition of `noMoreThanFive` which should be a list of all pairs `(x,y)` such that `x` is a number from 0 to 5 and `y` is also a number from 0 to 5, unless $x + y > 5$: `[(0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0), ...]`.

Ex37 Define `noMoreThanFive'` which should produce the same list as `noMoreThanFive`, but not use any inequality operators. Note that the type of `noMoreThanFive'` no longer has an `Ord` constraint on the type variable `a` so that you will receive a type error if you try to use any inequality operators.

Finally, a list comprehension may contain predicates to determine which elements produced by generators should be used for elements in the resulting list. For example, the following list comprehension generates all numbers which are factors of some number `n`. The `mod` function from the standard library calculates the remainder of two numbers:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], mod n x == 0]
```

Ex38 Complete the definition of `evens` which should be the list of all numbers from 0 to 100 which are even.

Ex39 Complete the definition of `multiples` which, given some natural number `n`, should produce the list of all factors of 3 and 5 in the range from 0 to `n` (inclusive). For example, `multiples 10` should evaluate to `[0,3,5,6,9,10]`.

Ex40 Finally, ensure that all your definitions are correct by running all of the unit tests with `stack test` in a terminal.

4.8 Recursive functions

Topics: Recursive functions.

These exercises are about recursive functions in Haskell. You can obtain the skeleton code by cloning the repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-recursive-functions
```

In a nutshell, recursive functions are functions which are defined in terms of themselves. They are used extensively in functional programming to repeatedly perform computations based on the arguments given to them. Most of the functions you will write in Haskell will be recursive.

📕 *Recommended reading*: Chapter 5 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 6 of *Programming in Haskell* (Hutton, 2016).

Ex41 Using explicit recursion, complete the definition of

```haskell
elem :: Eq a => a -> [a] -> Bool
```

which should determine whether some value of type `a` is contained in a list of values of type `a`. For example, `elem 4 [4,8,15,4]` should evaluate to `True` and `elem 7 [4,8,15,4]` should evaluate to `False`.

Ex42 Using explicit recursion, complete the definition of

```haskell
maximum :: Ord a => [a] -> a
```

which should find the greatest element of the list given as argument. For example, `maximum [1,2,3,2,1]` should evaluate to `3`. You may assume that `maximum` will never be called with the empty list so you do not need to define an equation for that case.

Ex43 Complete the definition of

```haskell
intersperse :: a -> [a] -> [a]
```

which should separate elements of a list with some separator of the same type as the elements of the list. For example, `intersperse '|' "CAKE"` should evaluate to `"C|A|K|E"`.

Hint: you may find it useful to define an additional function to help you.

Ex44 Complete the definition of

```haskell
subsequences :: [a] -> [[a]]
```

which should find all possible subsequences of the argument. For example, evaluating `subsequences "abc"` should result in a list such as `["", "a", "b", "c", "ab", "bc", "ac", "abc"]`. The order of the elements in the resulting list does not matter.

4.9 Higher-order functions

Topics: Higher-order functions and recursion schemes.

These exercises is about higher-order functions in Haskell. You can obtain the skeleton code by cloning the repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-higher-order-functions
```

In a nutshell, higher-order functions are functions which take other functions as arguments or return functions.

Recommended reading: Chapter 6 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 7 of *Programming in Haskell* (Hutton, 2016).

Ex45 For each of the following statements, discuss with someone (friend, tutor, rubber duck, etc.) whether it is true or false:

1. A function of type `a -> b -> c` returns a function.
2. A function of type `(a -> b) -> Int` returns a function.
3. A function of type `(Int, Bool) -> Char` is higher-order.
4. A function of type `a -> a` can be a higher-order function.

You have already learnt how to write functions using *explicit recursion*, as demonstrated in the definition of `and` below:

```haskell
and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

I.e. a function is explicitly recursive if it mentions itself by name in its definition: in this case, `and` is mentioned on the RHS of the second equation.

You have now also learnt how to use higher-order functions to abstract over common patterns. For example, you could define `and` with the help of `foldr` as the following, in which case the recursion is not explicit:

```haskell
and :: [Bool] -> Bool
and = foldr (&&) True
```

Ex46 Define `elem` entirely in terms of the following standard library functions: `not`, `null`, `filter` with an appropriate predicate, and function composition.

Ex47 Now define `elem` in terms of `foldr`. Do you find this definition clearer than the previous one?

Ex48 The `foldr1 :: (a -> a -> a) -> [a] -> a` function behaves like `foldr`, except that it assumes the list argument always has at least one element. It therefore only requires a binary operation and the list as arguments. Implement `maximum` using `foldr1`.

Ex49 Complete the definition of `any`, which should test whether at least one element of the list given as argument satisfies the predicate. For example, `any even [1,2,3]` should evaluate to `True`.

Ex50 Can you define `elem` using only `any` with an appropriate predicate?

Ex51 Complete the definition of `all`, which should test whether all elements of the list given as argument satisfy the predicate. For example, `all odd [1,3,5]` should evaluate to `True`.

Ex52 Implement the `flip` function which, given a function with two parameters as argument, produces a function with the parameters flipped. Note: there are no unit tests for this function, but your implementation will be correct if it is well typed according to the typing in the skeleton code.

Ex53 Complete the definition of

```haskell
takeWhile :: (a -> Bool) -> [a] -> [a]
```

which generalises the `take` function to a predicate: `takeWhile` should take elements from the list argument and return them while the predicate holds. For example, `takeWhile (< 3) [1,2,3,2,1]` should evaluate to `[1,2]`.

Ex54 Complete the definition of `zipWith`, which generalises `zip`. While `zip` puts elements from two lists into pairs in a resulting list, `zipWith` requires a function of type `a -> b -> c` as argument which combines elements of type `a` from the first list and elements of type `b` from the second list into elements of type `c` for the resulting list. For example, `zipWith replicate [1,2,3] ['a', 'b', 'c']` should evaluate to `["a", "bb", "ccc"]`.

Ex55 Complete the definition of `groupBy`, which should group elements of a list according to some predicate. For example, `groupBy (==) [1,2,2,3,4,4,1]` should evaluate to `[[1], [2,2], [3], [4,4], [1]]`.

Ex56 Complete the definition of

```
permutations :: Eq a => [a] -> [[a]]
```

which should find all possible permutations of the argument. For example, evaluating `permutations "abc"` should result in a list such as `["abc", "acb", "bac", "bca", "cab", "cba"]`. The order of the elements in the resulting list does not matter.

Ex57 Could you implement `permutations` without the *`Eq`* constraint on `a`?

4.10 Using other libraries

Topics: Dependency management, using definitions from arbitrary libraries

In Section 4.4, we learnt how to create our own package and that Haskell's standard library, the `Prelude` module, is contained in the `base` package which we typically have as a dependency by default. When building real software, it is not unusual to have a significant number of dependencies, not just one. After all, it does not make sense for a team of software engineers to build everything from scratch – instead it is a good idea to see what libraries other people have built already, there *may*[3] be more mature implementations of what you need already so there may be no need to invest time into building it again.

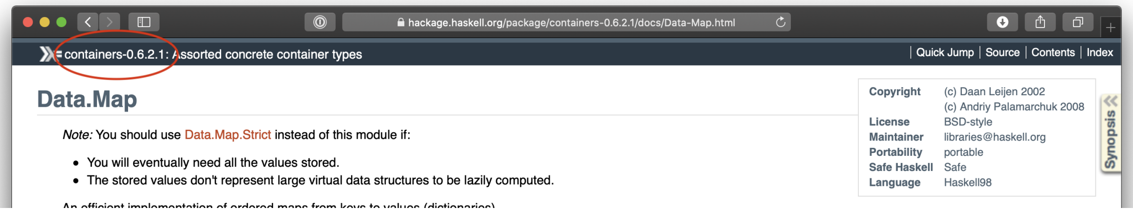
Haskell has a package repository called Hackage which you can search for Haskell packages:

`https://hackage.haskell.org/packages/search`

Alternatively to searching Hackage directly, you may also use Hoogle to search for functions and types across all indexed packages:

`https://hoogle.haskell.org`

When looking at the documentation for a module on Hackage, you can always see the name of the package (and the version) in the top left corner:


hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map.html
containers-0.6.2.1: Assorted concrete container types
Quick Jump | Source | Contents | Index
Synopsis

Data.Map

Note: You should use Data.Map.Strict instead of this module if:

- You will eventually need all the values stored.
- The stored values don't represent large virtual data structures to be lazily computed.

Copyright	(c) Daan Leijen 2002 (c) Andriy Palamarchuk 2008
License	BSD-style
Maintainer	libraries@haskell.org
Portability	portable
Safe Haskell	Safe
Language	Haskell98

I.e. to use the `Data.Map` module, you must add the `containers` package as a dependency to your project.

📘 *Recommended reading*: Chapter 7 of *Learn you a Haskell* (Lipovača, 2011).

Stack resolver Dependency management is actually quite a difficult problem because as we add more dependencies to a project, those dependencies may in turn have more dependencies, including with each other. This results in a *dependency*

[3]Whenever you add an external dependency to a project you work on, you should use your judgement it is a good idea to do so in terms of the library's maturity, whether it is being maintained, and for possible security implications.

graph where each edge is constrained by the version range that a package will accept for a particular dependency. In addition to being computationally difficult to solve all constraints, it is very easy for this to lead to unsatisfiable constraints if *e.g.* multiple packages depend on different versions of another package. To improve on this situation, the `stack` tool has a notion of "resolvers" – subsets of the Hackage package index in which all packages are known to be compatible with each other.

Ex58 For this exercise, we will be creating another simple package from scratch with the help of `stack`. In a terminal, run the following:

```
$ stack new lab-dependencies simple-library --resolver=lts-16.27
```

Ex59 Edit `lab-dependencies.cabal` to add a dependency to the `containers` package.

```
library
  -- ...
  build-depends:       base >= 4.7 && < 5, containers
```

With this dependency added, you can now import any of the modules from the `containers` package:

`https://hackage.haskell.org/package/containers`

The `containers` package contains implementations of some data structures, including maps (dictionaries) and sets.

Ex60 Modify `src/Lib.hs` to import the *`Data.Set`* module, qualified as *`S`* to avoid clashes with names of functions in *`Prelude`*:

```haskell
import qualified Data.Set as S
```

Ex61 Add a definition to test that you can use functions and types imported from *`Data.Set`*:

```haskell
set :: S.Set Int
set = S.union (S.fromList [0..5]) (S.fromList [3..10])
```

You may also wish to add `set` to the export list for the module (or remove the export list entirely to export everything automatically):

```haskell
module Lib (someFunc, set) where
```

Open the REPL with `stack repl` and evaluate `set` to check that the list you get as a result only contains every number once.

4.11 Data types

Topics: Algebraic data types, pattern matching, type parameters, type class instances, records.

These exercises are about algebraic data types in Haskell. You can obtain the skeleton code by cloning the repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-data-types
```

Algebraic data types (or just data types for short) are the primary way for us to define our own types.

Recommended reading: Chapter 8 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 8.1 of *Programming in Haskell* (Hutton, 2016).

A fairly simple example of an algebraic data type is the following:

```haskell
data IntPos = IntPos Int Int
```

This type, named `IntPos`, has a single constructor, also named `IntPos` with two parameters of type `Int`. The purpose of this type is to represent 2-dimensional positions.

Ex62 What is the type of the `IntPos` constructor? Use the REPL and `:t IntPos` to check your answer.

Ex63 It would be useful to test whether two values of type `IntPos` are the same. Complete the instance of the `Eq` type class for `IntPos`. Ensure that your implementation is correct by checking that both tests for it pass.

Ex64 It would also be useful to be able to turn values of type `IntPos` into corresponding `String` representations so that we can show them in the REPL. Complete the `Show` instance for `IntPos`. For example:

```
*Lab Lab> show (IntPos 4 15)
"IntPos 4 15"
```

Ex65 Complete the definition of `zeroPos`, which should represent the position where both coordinates are 0.

Ex66 Complete the definitions of the `x` and `y` functions, which should extract the first and second coordinate from an `IntPos` value, respectively. For example:

```
*Lab Lab> y (IntPos 4 15)
15
```

Often it is useful to define types as generically as possible. While our `IntPos` type is fine if we only ever want to represent positions using integer coordinates, we would have to duplicate a lot of code if we ever wanted to *e.g.* represent positions using floating point numbers as coordinates.

```haskell
data Pos a = Pos a a
```

Ex67 What is the type of the `Pos` constructor? Use the REPL and `:t Pos` to check your answer.

Ex68 It would be useful to test whether two values of type `Pos` are the same. Complete the instance of the `Eq` type class for `Pos`. Ensure that your implementation is correct by checking that both tests for it pass.

Ex69 It would also be useful to be able to turn values of type `Pos` into corresponding `String` representations so that we can show them in the REPL. Complete the `Show` instance for `Pos`. For example:

```
*Lab Lab> show (Pos 4.2 15.16)
"Pos 4.2 15.16"
```

Ex70 Complete the definition of `zero`, which should represent the position where both coordinates are 0.

Ex71 Complete the definitions of the `left` and `top` functions, which should extract the first and second coordinate from an `Pos` value, respectively. For example:

```
*Lab Lab> top (Pos 4.2 15.16)
15.16
```

Ensure that the tests for `left` and `top` succeed before proceeding.

Ex72 The Haskell compiler can construct *projection functions* like `left` and `top` which extract individual components from a data constructor automatically. Remove the definitions of `left` and `top` and change the definition of `Pos` to:

```haskell
data Pos a = Pos { left :: a, top :: a }
```

If you run the tests, you should find that the tests for `left` and `top` still succeed.

Ex73 It is important to remember that there are virtually no restrictions on the types of things that we can store in data constructors or that type variables can be instantiated with. For example, the following is a valid use of the `Pos` type we defined:

```
*Lab Lab> :t Pos (\x -> True) odd
Pos (\x -> True) odd :: Integral p => Pos (p -> Bool)
```

I.e. functions can be stored inside of data constructors. To practice this, we have defined the following data type:

```
data DocumentItem = ListItem (Int -> String)

doc :: [DocumentItem]
doc =
  [ ListItem (\n -> show n ++ ". An item")
  , ListItem (\n -> concat ["I am item #", show n])
  , ListItem (\n -> concat ["There. Are. ", show n, ". Items." ])
  ]
```

What is the type of `ListItem`? Verify your answer in the REPL.

Ex74 Complete the definition of the `render` function so that it produces the following results:

```
*Lab Lab> render [] 1
""

*Lab Lab> render (tail doc) 1
"I am item #1\nThere. Are. 2. Items.\n"

*Lab Lab> render doc 1
"1. An item\nI am item #2\nThere. Are. 3. Items.\n"
```

4.12 Recursive data types

Topics: Algebraic data types, binary trees, and red-black trees.

These exercises are about recursive data types in Haskell. You can obtain the skeleton code by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-recursive-types
```

Note that the skeleton code for these exercises will not initially compile and you need to complete some of the exercises first before it does.

Some of the unit tests simultaneously test several of your functions, such as *e.g.* to ensure that your trees have the correct depth after several insertions – a test failure could be due to an error in any one of the tested functions.

You may wish to read your *CS126 Design of Information Structures* notes or equivalent in conjunction with these exercises.

Further reading: if you are interested in the implementation of data structures in purely functional programming languages, *Purely functional data structures* (Okasaki, 1999) is an excellent book.

In Haskell, binary trees may be represented using the following data type:

```haskell
data BinaryTree a = Leaf | Node (BinaryTree a) a (BinaryTree a)
```

We parametrise the `BinaryTree` type over some type `a` to allow it to store values of some arbitrary type. Nodes contain a left subtree of type `BinaryTree a`, a value of type `a`, and a right subtree also of type `BinaryTree a`. An empty binary tree can be represented as follows:

```haskell
empty :: BinaryTree a
empty = Leaf
```

A particular variant of a binary tree is a *binary search tree*. Elements are inserted into branches of the tree according to some partial ordering relation. We can define a function to insert values into a binary search tree:

```haskell
insert :: Ord a => BinaryTree a -> a ->  BinaryTree a
insert Leaf x = Node Leaf x Leaf
insert (Node l y r) x
    | x < y     = Node (insert l x) y r
    | x > y     = Node l y (insert r x)
    | otherwise = Node l y r
```

If the binary tree is empty, we create a node for the value x we are trying to insert whose subtrees are empty. If the binary tree is a node, we compare the value x we are trying to insert with the value of the node y. If x is less than y, we insert it into the left subtree. If x is greater than y, we insert it into the right subtree. If x and y are equal, then the tree already contains the value and we don't do anything. In other words, our binary trees represent a set where each value may only appear once.

However, this implementation of binary search trees is not *balanced*. If we repeatedly insert values that are *e.g.* increasing, then we essentially end up with a linked list:

```
   foldl insert empty [1..5]
=> Node Leaf 1 (Node Leaf 2 (Node Leaf 3 (
          Node Leaf 4 (Node Leaf 5))))
```

Looking up an element therefore has a worst-case time complexity of $\mathcal{O}(n)$ for trees with n elements. We can do better. *Red-black trees* are a way of keeping binary trees balanced so that the depth of all branches in the tree is roughly the same. If it is then used as a search tree, looking up elements takes at most $\mathcal{O}(\log n)$ time.

Ex75 Complete the definition of the `Colour` data type by replacing the `???` with suitable data constructors. The `Colour` type should represent either the colour `Red` or the colour `Black`.

Ex76 Complete the `Show` instance for the `Colour` type so that `show Red` evaluates to `"Red"` and `show Black` evaluates to `"Black"`.

Ex77 Complete the definition of the `Tree` data type by replacing the `???` with a suitable data constructor for `Node`s. The `Tree` type should represent red-black trees. The `Leaf` constructor is already given to you. You need to add one more constructors for `Node`s, which consist of a colour, left subtree, value, and a right subtree.

Ex78 Complete the definition of `empty`, which should represent an empty red-black tree.

Ex79 Complete the definition of the `singleton` function which takes some element of type a and constructs a red-black tree containing only that element. Use `Red` as the colour. For example, `singleton 5` should evaluate to `Node Red Leaf 5 Leaf` or equivalent depending on your definition of the `Node` constructor.

Ex80 Complete the definition of the `makeBlack` function which, given some arbitrary red-black tree, should change the colour of the root node to `Black` regardless of its current colour.

Ex81 Complete the definition of the `depth` function which, given some arbitrary red-black tree, should calculate the depth of the tree. An empty red-black tree should have depth 0 so that `depth Leaf` evaluates to `0`.

Ex82 Implement the `toList` function which should convert a red-black tree to a list through inorder traversal.

Ex83 Complete the definition of the `member` function, which given some value of type a and a red-black tree, should determine whether the value is contained in the tree.

Ex84 Complete the definition of the `balance` function, which given some arbitrary red-black tree, should balance it according to the transformations shown in Figure 4.1.

You can accomplish this simply by pattern-matching on the tree given as input to determine whether it is unbalanced and returning a balanced version of it if necessary. One of the four transformations is already implemented for you.

Ex85 Complete the definition of the `insert` function, which should insert some value of type a into a red-black tree. New nodes in the tree should initially be red. Use the `balance` function in the appropriate places to ensure that the resulting tree is balanced. The root of a red-black tree should always be black. For example, if you insert a bunch of elements from a list into a tree, the result in the REPL might look something like this:

```
Lab> foldl insert empty []
Leaf

Lab> foldl insert empty [42]
Node Black Leaf 42 Leaf

Lab> foldl insert empty ["CS141", "CS263"]
Node Black Leaf "CS141" (Node Red Leaf "CS263" Leaf)

Lab> foldl insert empty "KOAN"
Node Black
    (Node Black (Node Red Leaf 'A' Leaf) 'K' Leaf)
    'N'
    (Node Black Leaf 'O' Leaf)

Lab> foldl insert empty "WITTER"
Node Black
    (Node Black (Node Red Leaf 'E' Leaf)
    'I'
    (Node Red Leaf 'R' Leaf)) 'T' (Node Black Leaf 'W' Leaf)
```

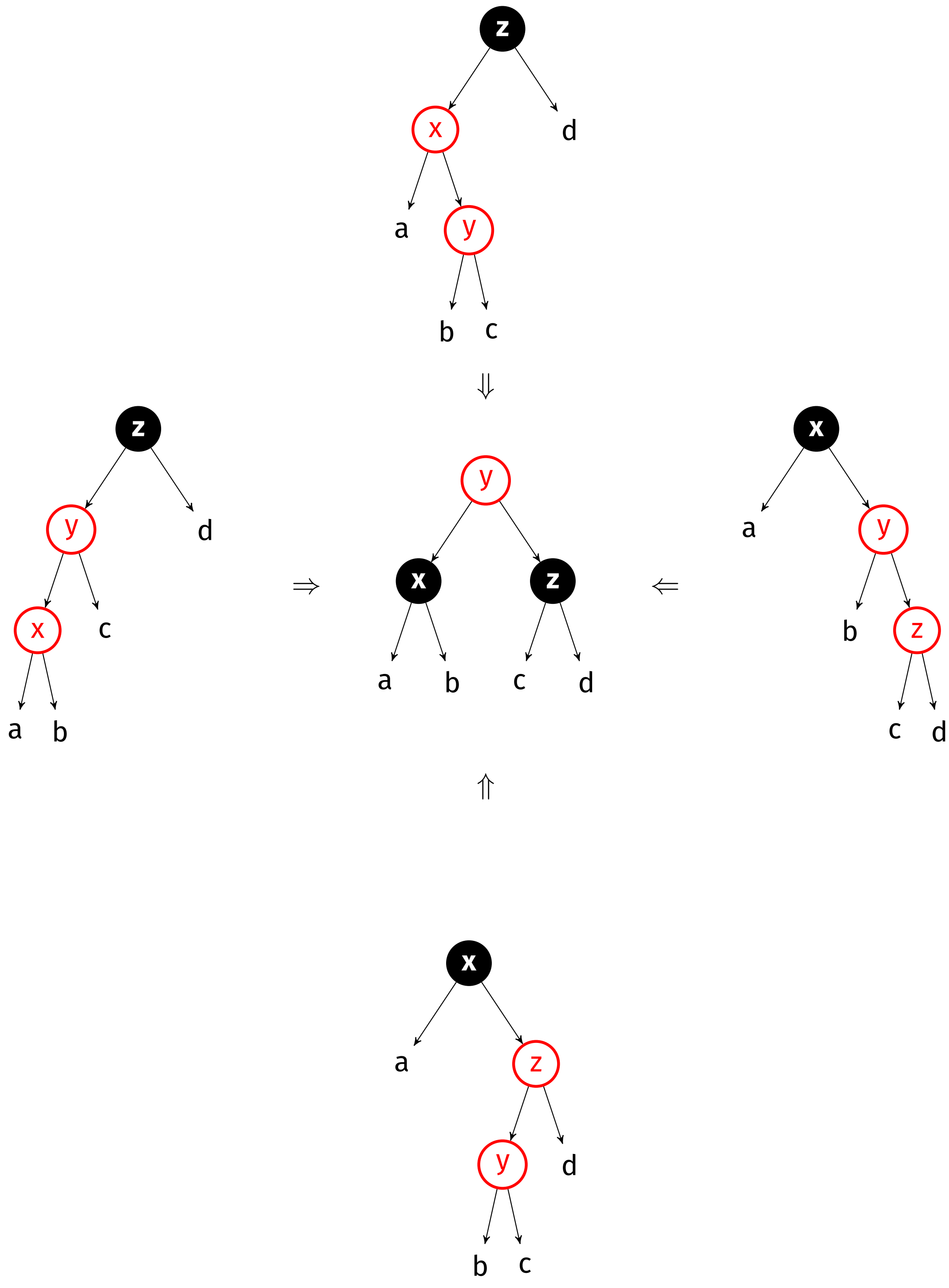


Figure 4.1: Balancing red-black trees

The exact result depends on the names you chose for your constructors and the order of their parameters. There are two unit tests for this lab which check that the depth of your red-black trees never exceed $2 \times \lfloor \log(n+1) \rfloor$ where n is the number of nodes in the tree and that converting a red-black tree to a list results in a sorted list.

Ex86 There are two invariants which red-black trees should never violate:

1. No red node has a red child.
2. Every path from the root to a leaf contains the same number of black nodes.

Convince yourself that these invariants are not violated by your implementation.

4.13 Lazy evaluation

Topics: Lazy evaluation, infinite data structures.

These exercises are about lazy evaluation and infinite data structures. You can obtain the skeleton code by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-lazy-evaluation
```

Recommended reading: Chapter 15 of *Programming in Haskell* (Hutton, 2016).

Further reading: if you want to read the full, gory technical details on how lazy evaluation is implemented by Haskell's runtime system, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine* (Peyton Jones, 1992) is for you.

Infinite data structures

In the lecture on lazy evaluation, you saw that Haskell supports infinite data structures such as infinite lists. This is possible because, at runtime, variables in Haskell are just pointers to closures. For example, we saw the following definition in the lecture:

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
```

Recall that the Haskell compiler transforms expressions which appear as arguments to functions into let-bound definitions:

```haskell
from :: Int -> [Int]
from n = let ns = from (n+1) in n : ns
```

Thus, when `from n` is called for some `n`, a new closure for `ns` is allocated on the heap. Because of lazy evaluation, the call to `from (n+1)` is not evaluated immediately. It is only evaluated when the tail of `n na: ns` is needed and then the closure represented by `ns` is updated with the result of `from (n+1)`. The call to `from (n+1)` will allocate yet another closure for `from (n+2)` which is only evaluated when the tail of the tail of `from n` is needed.

Ex87 Complete the definition of `ones` which should represent an infinite list where all elements are `1`.

You also saw that the following definition for the infinite list of Fibonacci numbers can be implemented elegantly and efficiently as the following in Haskell:

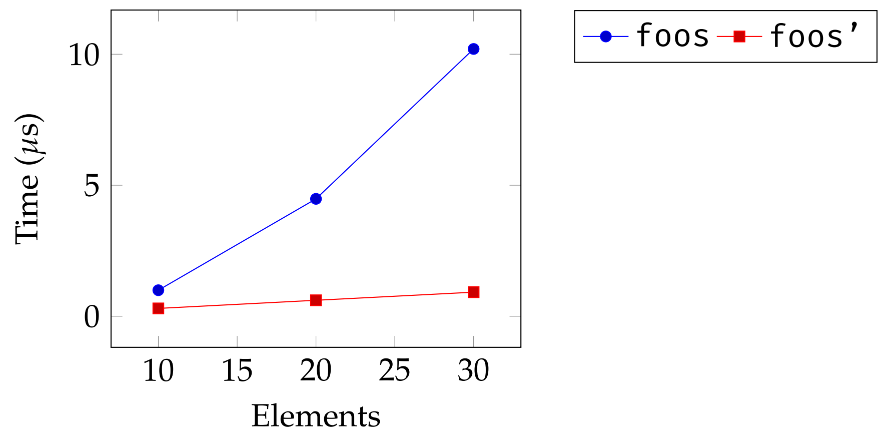


Figure 4.2: Time complexities of `foos` and `foos'`

```haskell
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

The infinite list represented by `fibs` can be generated in linear time for the number of elements requested. This is possible because `fibs` is transformed into the following:

```haskell
fibs :: [Integer]
fibs = let xs = tail fibs
           ys = zipWith (+) fibs xs
           zs = 1 : ys
       in 1 : zs
```

All closures involved in this definition can be updated with their results. Therefore, `fibs` refers to a cons cell (a kind of closure) where the head is `1` and the closure pointed to by `zs` is the tail. The closure represented by `zs` is also a cons cell where the head is `1` and the tail is pointed to by `ys`. The closure represented by `ys` will be updated with the result of `zipWith (+) fibs xs` when more than two elements are requested from `fibs`, *i.e.* the tail of `zs` is inspected by a `case`-expression somewhere. The `zipWith` function, in turn, allocates more closures when called, thus recursively adding more cons cells as needed.

However, not all closures can be updated. Suppose we want to generalise the definition of `fibs` to sequences with arbitrary seed values `x` and `y`:

```haskell
foos :: Integer -> Integer -> [Integer]
foos x y = x : y : zipWith (+) (foos x y) (tail (foos x y))
```

This will be exponentially slow, because Haskell will not update the closure for `foos` with the list that is being generated. As a result, every call to `foos x y` will result in the list being generated from the beginning. This makes sense, because `foos` is parametrised over `x` and `y` and it would be wrong to update the closure for `foos` with the list generated for some specific `x` and `y`.

Ex88 Implement `foos'` to produce the same sequence as `foos`, but in linear time. *Hint*: you need to find a way to allocate a closure which is specific to some `x` and `y` and can be updated.

Ex89 You can run `stack bench` to compare the time complexities of `foos` and `foos'`. If you have done everything right, you should see a bar graph with similar growth as the graph shown in Figure 4.2 in the `benchmark.html` file which is generated by `stack bench`.

4.14 Equational reasoning

Topics: Equational reasoning, constructive induction.

Haskell is a purely functional programming language, which allows us to easily prove formal properties about our functions. This is nice because it means we do not have to rely on testing (which is not exhaustive) and can instead use structural induction to exhaustively cover all possible inputs to a function. The approach to proving properties we use for Haskell programs is called *equational reasoning* – this approach works by starting with an equation and rewriting both sides of it until they are the same.

Recommended reading: Chapters 16 and 17 of *Programming in Haskell* (Hutton, 2016).

If you are looking for more exercises, have a look at the past exam papers – Question 4 on each paper has more theorems to prove.

Induction on natural numbers

Recall that we have already proved the following (monoidal) properties about natural numbers in the lecture on equational reasoning (Section 3.1):

Left unit	$add\ Z\ x$	$=$	x
Right unit	$add\ x\ Z$	$=$	x
Associativity	$add\ x\ (add\ y\ z)$	$=$	$add\ (add\ x\ y)\ z$

Ex90 Prove the following property about addition just by rewriting one side of the equation until you end up with the other side:

$$\forall n :: Nat. S\ n = add\ (S\ Z)\ n$$

You should show each step of the proof along with a comment to say what you have done at that particular step, as shown in the lecture and the proofs in Section 3.1.

Ex91 Prove the following property about addition by induction on n. For this proof, you will need to make use of some of the other properties you know about *add*, including the one you proved for Exercise 90.

$$\forall n\ m :: Nat. add\ (S\ n)\ m = add\ n\ (S\ m)$$

Ex92 Finally, using all the properties you know about *add* so far, prove that *add* commutes:

$$\forall n\ m :: Nat. add\ n\ m = add\ m\ n$$

Induction on lists

In Haskell, the `reverse` function can be defined as follows:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This makes use of the `(++)` operator, which is defined as follows:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Ex93 Prove that $(+\!\!+)$ has a left identity by rewriting the following equation:

$$\forall xs :: [a] \;.\quad [\,] +\!\!+ xs = xs$$

Ex94 Prove that $(+\!\!+)$ has a right identity by induction on xs:

$$\forall xs :: [a] \;.\quad xs +\!\!+ [\,] = xs$$

Ex95 Prove that $(+\!\!+)$ is associative by induction on xs:

$$\forall xs\ ys\ zs :: [a] \;.\quad xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Ex96 Prove that *reverse* preserves singleton lists by rewriting the following equation:

$$\forall x :: a \;.\quad reverse\ [x] = [x]$$

Ex97 Prove that *reverse* distributes over $(+\!\!+)$ by induction on xs. You will need some of the properties you have proved so far about $(+\!\!+)$ and *reverse*.

$$\forall xs\ ys :: [a] \;.\quad reverse\ (xs +\!\!+ ys) = reverse\ ys +\!\!+ reverse\ xs$$

Ex98 Prove the following property about *reverse* by induction on xs. You will need some of the properties you have proved so far about $(+\!\!+)$ and *reverse*.

$$\forall xs :: [a] \;.\quad reverse\ (reverse\ xs) = xs$$

Constructive induction

It is possible to use induction to *calculate* faster function definitions. This is referred to as *constructive induction*. For example, consider our current definition of `reverse`:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This definition is inefficient because the `(++)` operator runs in $\mathcal{O}(n)$ time where n is the length of the first argument. The `reverse` function runs in quadratic time as a result. We can do better by combining the behaviour of `reverse` and `(++)` into one new function which does both. We begin by expressing this idea as the following specification:

```
rev :: [a] -> [a] -> [a]
rev xs ys = reverse xs ++ ys
```

The `rev` function takes two lists as arguments, reverses the first and then appends the second list to it. Our goal is now to use induction to come up with a new definition for `rev` which neither uses `reverse` nor `(++)`. We do this by taking the current definition and performing induction on `xs`. Recall that there are two cases for induction on lists: the empty list (a base case) and cons (a recursive case). We can write down a skeleton for the new definition of `rev` by covering these two cases:

```
rev :: [a] -> [a] -> [a]
rev []     ys = ???
rev (x:xs) ys = ???
```

Ex99 Replace the `???` in the first equation by reducing `reverse [] ++ ys` as much as possible. The resulting expression should neither contain `reverse` nor `(++)`. *Hint*: you only need to apply `reverse` and `(++)` until you are left with an expression which cannot be reduced any further. This expression can then be used as the right-hand side of the first equation above.

Ex100 Replace the `???` in the second equation by reducing `reverse (x:xs) ++ ys` as much as possible. The resulting expression should neither contain `reverse` nor `(++)`. *Hint*: in this case, you can use the specification `rev xs ys = reverse xs ++ ys` as induction hypothesis.

4.15 Foldables

Topics: Foldables.

These exercises are about foldables, *i.e.* data structures for which we can implement `foldr`. As usual, you can obtain the skeleton code for this this lab by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-foldable
```

Recommended reading: Chapter 14 of *Programming in Haskell* (Hutton, 2016).

Ex101 Consider the following definition of a data type for a simple expression language consisting of variables, integer values, and addition:

```
data Expr a = Var a
            | Val Int
            | Add (Expr a) (Expr a)
```

This type is parametrised over some type a so that variables (the `Var` constructor) can be represented in different ways depending on what information needs to be associated with variables. This is useful if, for example, you are writing a compiler and want to initially only have variable names associated with your variables, but later add information such as types or line numbers. Some valid examples of `Expr` values are:

```
e1 = Var "x"
e2 = Var ("x",22)
e3 = Val 4
e4 = Add (Val 8) (Var "y")
e5 = Add (Val 8) (Var 7)
e6 = Add e2 (Var ("y",42))
e7 = Add e6 e6
```

Complete the `Foldable` instance for `Expr`. Once implemented, you should be able to do the following in the REPL:

```
toList e3 ==> []
toList e5 ==> [7]
sum e5    ==> 7
toList e7 ==> [("x",22),("y",42),("x",22),("y",42)]
length e7 ==> 4
```

Ex102 A useful data structure in purely functional programming languages is the *zipper*. Zippers can help make traversing data structures more efficient if repeated access to one element is required. Zippers can be defined for arbitrary data structures, but to keep things simple we will only look at lists. For lists, the zipper is defined as shown in the code snippet below:

```
data Zipper a = Zipper [a] a [a]
```

The zipper for lists is a bit like a Turing tape: we have an element of type a that is in the "view" and we have elements to the "left" of it (the first `[a]`) as well as to the "right" of it (the second `[a]`). We can convert ordinary, non-empty lists into zippers with a function of type:

```
fromList :: [a] -> Zipper a
```

This should work as follows: the head of the input list should be the element that is in the "view" and the tail of the input list should be to the "right" of the element in the "view". Implement this function now and ensure that the tests for it pass.

Ex103 There are three useful functions to work with zippers. Their type signatures are:

```
view  :: Zipper a -> a
left  :: Zipper a -> Zipper a
right :: Zipper a -> Zipper a
```

The `view` function simply retrieves the element that is in the "view". If the list of elements to the "right" of the "view" is not empty, the `left` function shifts elements to the "left" so that the "view" of the input is the first element on the "left" and that the first element of the "right" of the input is then in the "view". The `right` function does the opposite and shifts elements to the "right". Implement all three functions now and ensure that the tests pass.

Ex104 The `Zipper` type can be made an instance of `Foldable`. Complete the definition of `foldr` for it now. In essence, this should satisfy the following equations:

```
toList (fromList xs) == xs
foldr f z (fromList xs) == foldr f z xs
```

Ex105 As we have seen in the lectures, we can define very useful functions that work for any type provided that a `Foldable` instance exists. Generalise the `filter` function to a function `filterF` which should work on all types that have an instance of `Foldable`. Some examples of its usage are shown below:

```
filterF ((=="x") . fst) e7                ==> [("x",22),("x",22)]
filterF (>5) (Zipper [1,5,10] 8 [6,7,2]) ==> [10,8,6,7]
```

Ex106 Could you define an even more general `filter` function which can return the filtered elements in an arbitrary data structure that satisfies some type class constraints? In other words, a function with *e.g.* the following typing where `???` should be replaced by your type class constraints:

```
filterFA :: (Foldable f, ??? g) => (a -> Bool) -> f a -> g a
```

Ex107 We can write *even more* generic, useful functions by combining multiple type class constraints as follows (note that this uses the `Alternative` type class which is described in full in in Section 4.17):

```
asum :: (Alternative f, Foldable t) => t (f a) -> f a
```

Implement this function in terms of `foldr` and `(<|>)` so that combines the elements of some data structure with `(<|>)`. See below for examples of `asum` in action:

```
asum [Nothing, Just 4, Nothing]                    ==> Just 4
asum (Zipper [Just 4, Nothing] (Just 5) [])        ==> Just 4
asum (Zipper [Nothing, Nothing] (Just 5) [])       ==> Just 5
asum (Zipper [Nothing, Nothing] Nothing [Just 1]) ==> Just 1
```

4.16 Functors

Topics: Functors.

These exercises are about functors. We introduce you to a range of different types that are functors, in addition to those that you have seen in the lectures. Many of the types you will work with in these exercises may seem outright silly at this point. However, they we will continue to work with them in the coming weeks and you will see how they become increasingly more useful as we get to know more and more abstractions. As usual, you can obtain the skeleton code by cloning the respective repository from GitHub with the following command:

```
$ git clone https://github.com/fpclass/lab-functors
```

Recommended reading: Chapter 11 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 12.1 of *Programming in Haskell* (Hutton, 2016).

Ex108 We start off with one of the simplest types that is a functor: the `Identity` type. This type is defined as follows:

```
data Identity a = Identity a
```

This may seem like quite a silly type to define – what is it good for? We will find out later. For now, define an instance of `Functor` for it. Once defined, using `fmap` will have the same effect as just applying a function directly to the value that we put inside of `Identity`:

```
fmap (+2) (Identity 4)             ==> Identity 6
fmap length (Identity "Witter")    ==> Identity 6
fmap ($ "cake") (Identity length)  ==> Identity 4
fmap (map (=='o')) (Identity "foo")
==> Identity [False,True,True]
```

Ex109 Once you have implemented `fmap`, you may wish to verify that your instance of `Functor` for `Identity` obeys the functor laws. Running the tests with `stack test` will give you some indication, but you could also do it formally with some proofs by induction for the two functor laws.

Ex110 Another type that looks just as silly is the `Const` type which can be defined as:

```
data Const v a = Const v
```

This type is also a functor. Complete the `Functor` instance for it. Once implemented, you can test it's behaviour (or rather lack thereof):

```
fmap (+2) (Const 4)              ==> Const 4
fmap length (Const "Witter")     ==> Const "Witter"
fmap (map (=='o')) (Const "foo") ==> Const "foo"
```

As we can see, the "effect" of this functor is that it always preserves the values stored inside of it and the function we apply with `fmap` does nothing.

Ex111 You may again wish to convince yourself that your implementation of `fmap` for this type obeys the functor laws.

Ex112 You are given a definition for a type `Point` that can be used to represent points in two-dimensional space as well as other pairs of values of the same type:

```
data Point a = Point a a
```

For example, some values of this type are:

```
p1 :: Point Int
p1 = Point 4 5

p2 :: Point Double
p2 = Point 2.3 4.2

p3 :: Point String
p3 = Point "hello" "world"
```

The `Point` type is a functor. Complete the `Functor` instance for it. Some examples of what you should be able to do with the `Functor` instance for `Point` are:

```
fmap (+1) p1    ==> Point 5 6
fmap (+1) p2    ==> Point 3.3 5.2
fmap show p2    ==> Point "2.3" "4.2"
fmap length p3 ==> Point 5 5
```

As you can see, having `fmap` available for a type allows us to perform quite the range of operations on values of the `Point` type.

Ex113 You may once again wish to verify that your instance of `Functor` for `Point` obeys the functor laws.

Ex114 Consider the following definition of *rose trees*:

```
data RoseTree a = Leaf a | Node [RoseTree a]
```

In a rose tree, every node can have zero or more children. This type is also a functor. Complete the instance of `Functor` for `RoseTree`. Below are some examples of what is possible with `fmap` for `RoseTree`:

```
fmap (+5) (Leaf 4)                ==> Leaf 9
fmap (+5) (Node [])               ==> Node []
fmap (+5) (Node [Leaf 4, Leaf 8]) ==> Node [Leaf 9, Leaf 13]
fmap length (Node [Node [], Node [Leaf "duck"]])
==> Node [Node [], Node [Leaf 4]]
fmap sum (Node [Node [Leaf [1,2,3]], Node [Leaf [9,8,7]]])
==> Node [Node [Leaf 6], Node [Leaf 24]]
```

Ex115 A property of functors is that, given any two functors, they can be composed to form a new functor which combines both. This is useful if, for example, we want to exploit the fact that lists are functors to work more easily with nested lists. To begin, let us define a suitable type to represent a data structure where values of type `g a` are nested inside containers of type `f`:

```
data Compose f g a = Compose (f (g a))
```

What is the type of the `Compose` constructor?

Ex116 The `Compose` type is a functor. Complete the respective instance of `Functor`. Once implemented, you can very easily apply `fmap` to arbitrarily nested data structures:

```
fmap (+5) (Compose [[1,2,3], [4,5,6]])
==> Compose [[6,7,8], [9,10,11]]
fmap not (Compose [Just True, Just False, Nothing])
==> Compose [Just False, Just True, Nothing]
fmap even (Compose (Node [Leaf [1,2,3]]))
==> Compose (Node [Leaf [False, True, False]])
fmap (+5) (Compose (Compose [[[1,2],[3]], [[4],[5,6]]]))
==> Compose (Compose [[[6,7],[8]],[[9],[10,11]]])
```

Ex117 Proving that `fmap` for `Compose` obeys the functor laws is significantly more interesting than for the previous types. Can you do it? *Hint*: you need to assume that the two underlying functors obey the laws and make use of them.

Consider the following definition of a data type in Haskell:

```
data State s a = St (s -> (a,s))
```

This type, which we have named `State`, has two type parameters `s` and `a`. It has a single data constructor, named `St`, which has a single parameter of type `s -> (a,s)`. Therefore, the type of `St` is `(s -> (a,s)) -> State s a`. In other words, given a function of type `s -> (a,s)`, the `St` constructor produces a value of type `State s a`. Our intuition for this type is going to be that `St` is a wrapper around "stateful" functions: that is, functions which require some initial state (of some type `s`) as argument and return a pair consisting of a result (of some type `a`) and a potentially new state (of the same type `s` as the initial state). For example, we could define the following value of `State Int Int`:

```
fresh :: State Int Int
fresh = St (\s -> (s, s+1))
```

Let us also define a little helper function that is useful to have when working with the `State` type, which extracts the function `m` of type `s -> (a,s)` from a value of type `State s a` and applies it to some initial state `s` of type `s`:

```
runState :: State s a -> s -> (a, s)
runState (St m) s = m s
```

Using both of these definitions, we can then get some results:

```
runState fresh 4                                ==> (4,5)
runState fresh 7                                ==> (7,8)
let (x,s') = runState fresh 7 in runState fresh s' ==> (8,9)
```

While this is currently not overly interesting and a rather clunky, these definitions will provide the foundations for much more exciting things that are yet to come...

Ex118 Meanwhile, back to the grind: the `State` type is a functor. Complete the definition of `fmap` for it. Some examples of it in action:

```
runState (fmap (*2) fresh) 4 ==> (8,5)
runState (fmap show fresh) 7 ==> ("7",8)
```

Hint: implementing `fmap` for `State` is mostly a game with types. If you are unsure of what to write, place a typed hole `_` in the place where you do not know what expression to use and the Haskell compiler will tell you what the type of the expression should be, what variables are in scope and what their types are.

Ex119 We know that, in Haskell, functions have types of the form `a -> b` where `a` is the domain of the function (*i.e.* the type of arguments that the function accepts) and `b` is the co-domain (*i.e.* the type of values the function produces). In Haskell, type constructors can be partially applied just like functions. For example, `(->) a`, is the function type constructor `(->)` applied to only one argument `a`. Of course there are no values of this type, so why is this useful? Well, it turns out that this type is a functor! Can you complete the definition of `Functor` for it and figure out what `fmap` does for this type?

```
instance Functor ((->) r) where
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap = undefined
```

Hint: this is very easy if you recognise the type of `fmap` where `f` is replaced by `(->) r` (shown in the comment above) from somewhere...

4.17 Applicative functors

Topics: Applicative functors, parsing.

These exercises are about using applicative functors. As usual, you can obtain the skeleton code for this this lab by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-applicatives
```

Recommended reading: Chapter 11 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 12.2 of *Programming in Haskell* (Hutton, 2016).

The aim of this lab is to implement a small library for constructing *parsers* as well as to build a parser for a small expression language. At the end of this lab, we will have a function called `parseAndEval` which, given a string representation of simple arithmetic expressions, will be able to convert them into a value of an `Expr` type which is then interpreted. The `parseAndEval` function may fail if the input string is not well formatted. Here are a few examples of how it will work:

```
parseAndEval "4"                     ==> Just 4
parseAndEval "(15 + 16)"             ==> Just 31
parseAndEval "((2 + 4) + (9 + 2))" ==> Just 17
parseAndEval "(2"                    ==> Nothing
```

Parsers

A parser is a program which, given some text as input, attempts to convert it into a more structured representation in memory. For example, a parser might be part of a compiler where it converts code written in a particular programming language into a representation that the compiler can work with, such as a type like `Expr` that we have seen in the lectures. We can therefore think of parsers as functions of the following type:

```
String -> Maybe (a, String)
```

That is, given a value of type `String` as input, a parser consumes some of the input and tries to return a value of some type `a` (the structured representation) as well as all of the remaining input. However, it may fail and return `Nothing` if the input does not contain something in the format expected by a particular parser. One approach to constructing parsers is called *parser combinators*. The idea is to have a library of very basic parsers, such as one which parses a single character or one which allows choice between two different parsers, and combine them to construct more meaningful parsers. With this in mind, we define a type of parser computations as an algebraic data type. This will ultimately allow us to make the input to individual parsers implicit:

```
data Parser a = MkParser (String -> Maybe (a, String))
```

A value of type `Parser a` represents a parser computation which returns a value of `a`. The `MkParser` data constructor has type `(String -> Maybe (a, String)) -> Parser a`. That is, given some function which implements the parsing behaviour, a value of type `Parser a` is returned. You are given a function

```
parse :: Parser a -> String -> Maybe a
```

which, given a computation of type `Parser a` and an input `String`, may return some value of type `a` wrapped into `Just` or `Nothing` if parsing failed.

Ex120 Implement the `ch` function which, given a predicate on `Char` values, should construct a parser which inspects the first character in its input: if the first character satisfies the predicate, then the character should be returned together with the remaining input. If the first character does not satisfy the predicate or the input is empty, then the parser should return `Nothing`:

```
parse (ch (=='x')) ""    ==> Nothing
parse (ch (=='x')) "yzx" ==> Nothing
parse (ch (=='x')) "xyz" ==> Just ('x',"yz")
```

Ex121 The `Parser` type is a functor where we can apply functions of type `a -> b` to the result of the parser. Complete the `Functor` instance for `Parser` by replacing `undefined` with suitable code:

```
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f (MkParser p) = undefined
```

Once implemented, you should be able to run the following expressions in the REPL to get the results shown:

```
parse (fmap isUpper (ch (=='x'))) "xyz"    ==> Just (False,"yz")
parse (fmap isUpper (ch (=='y'))) "xyz"    ==> Nothing
parse (fmap digitToInt (ch isDigit)) "1xy" ==> Just (1,"xy")
parse (fmap digitToInt (ch isDigit)) "xy"  ==> Nothing
```

Ex122 The `Parser` type is also an applicative functor. The `pure` function should construct a parser which does not touch its input and always returns the argument of type `a` that is given to `pure`:

```
instance Applicative Parser where
    -- pure :: a -> Parser a
    pure x = undefined
```

Once you have implemented `pure`, the following expressions should evaluate to the results shown:

```
parse (pure True) "xyz" ==> Just (True, "xyz")
parse (pure True) ""    ==> Just (True, "")
parse (pure 42) "123"   ==> Just (42, "123")
```

Ex123 The `(<*>)` operator should compose two parsers into one parser. For `Parser`, it has the following type:

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

That is, given two parsers where one returns a function of type `a -> b` and one returns a value of type `a`, it should construct a parser which combines the two parsers given as arguments into one parser that returns a value of type `b`. You are given the following skeleton:

```
(MkParser a) <*> p = MkParser $ \xs -> case a xs of
    Nothing      -> undefined
    Just (f, ys) -> let (MkParser b) = p in case b ys of
        Nothing      -> undefined
        Just (x, zs) -> undefined
```

The intuition here is that `(<*>)` returns a computation of type `Parser b`, *i.e.* a parser computation which returns a value of type `b`. This computation is defined using the `MkParser` constructor applied to a parsing function, which takes a string `xs` as input, then applies the parsing function `a`, obtained from pattern-matching on the computation of type `Parser (a -> b)`, to it. This either results in failure, in which case the whole thing you should fail (you need to fill in this behaviour for the first `undefined`). If `a xs` is successful, then the result is a function `f :: a -> b` and the remaining input `ys :: String`. We then pattern-match on `p :: Parser a` to obtain the second parsing function `b :: String -> Maybe (a, String)` which we then apply to the remaining input `ys` obtained from evaluating the first parsing function. Applying `b` to `ys` results in a computation of type `Maybe (a, String)` so we pattern-match on it to determine whether it failed or succeeded. If it failed, the whole thing should fail again (you need to implement this behaviour where the second `undefined` is). If the second parsing function succeeds, then we get a result `x :: a` and the remaining input `zs :: String`. You now need to replace the third `undefined` with a value of type `Maybe (b, String)` which should indicate success and contain the result of applying `f` to `x` alongside the remaining input of the second parsing function. Once you have implemented `(<*>)`, the following expressions should evaluate to the results shown:

```
parse (pure digitToInt <*> ch isDigit) "1yz"
==> Just (1,"yz")
parse (pure digitToInt <*> ch isDigit) "xyz"
==> Nothing
parse ((\x y -> [x,y]) <$> ch isDigit <*> ch isDigit) "123"
==> Just ("12","3")
parse ((\x y -> [x,y]) <$> ch isDigit <*> ch isDigit) "x23"
==> Nothing
parse ((\x y -> [x,y]) <$> ch isDigit <*> ch isDigit) "1x3"
==> Nothing
```

Alternatives

When working with applicative functors, it is sometimes useful to express the notion of choice. For example, in the case of parsers, if one parser returns failure, we may want to try a different parser to see if it succeeds instead. The *`Alternative`* type class provides some functions and operators to support this idea:

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a

    some  :: f a -> f [a]
    some p = (:) <$> p <*> many p

    many  :: f a -> f [a]
    many p = some p <|> pure []
```

Ex124 For the `Parser` type, the `empty` value has type `Parser a`. It should represent a computation which *always* fails. For example:

```
parse empty "xyz"                              ==> Nothing
parse (const <$> ch (const True) <*> empty) "xyz" ==> Nothing
```

As an aside, `ch (const True)` is a parser which will accept any character.

Ex125 The `(<|>)` operator has type `Parser a -> Parser a -> Parser a` for the `Alternative` instance for `Parser`. The intuition with this operator is that it will try the first parser. If the first parser succeeds, then its result is returned. Otherwise, the second parser is used and its result determines the overall result. For example:

```
parse (ch (=='x') <|> ch (=='y')) "x12" ==> Just ('x', "12")
parse (ch (=='x') <|> ch (=='y')) "y12" ==> Just ('y', "12")
parse (ch (=='x') <|> ch (=='y')) "z12" ==> Nothing
parse (ch (=='x') <|> ch isDigit) "y12" ==> Nothing
parse (ch (=='x') <|> ch isDigit) "x12" ==> Just ('x', "12")
parse (ch (=='x') <|> ch isDigit) "012" ==> Just ('0', "12")
```

Ex126 The `some` and `many` functions may be used to repeatedly invoke `p` and to generate a list of the results. While `some` should only succeed with the list of results obtained from `p` if `p` succeeds at least once, `many` should also return with the list of results obtained from `p` but may return the empty list if `p` does not even succeed once. You will have to implement either `some` or `many` as the default implementations are defined in terms of each other. For example:

```
parse (some (ch isDigit)) "xyz"    ==> Nothing
parse (many (ch isDigit)) "xyz"    ==> Just ("", "xyz")
parse (some (ch isDigit)) "1xyz"   ==> Just ("1", "xyz")
parse (many (ch isDigit)) "1xyz"   ==> Just ("1", "xyz")
parse (some (ch isDigit)) "123xyz" ==> Just ("123", "xyz")
parse (many (ch isDigit)) "123xyz" ==> Just ("123", "xyz")
```

Parser combinators

We can now define some basic parsers for common things we might wish to do. We will later be able to combine these basic parsers into more sophisticated parsers.

Ex127 Complete the definition of `token`, which should return the result of `p` which may be preceded by zero or more whitespace characters. The `whitespace :: Parser String` computation will be of use, which parses zero or more whitespace characters.

```
parse (token (ch (=='?'))) "?"     ==> Just ('?', "")
parse (token (ch (=='?'))) "    ?" ==> Just ('?', "")
```

Hint: The `(*>)` and `(<*)` from the lecture on applicative functors will be useful for this definition as well as coming ones.

Ex128 Complete the definition of `between`, which should return the result of `p`. `p` should be preceded by `open` and followed by `close`, the results of which should be discarded.

```
parse (between (ch (=='{')) (ch (=='}')) nat) "123}"
==> Nothing
parse (between (ch (=='{')) (ch (=='}')) nat) "{123"
==> Nothing
parse (between (ch (=='{')) (ch (=='}')) nat) "{123}"
==> Just (123,"")
```

Expression language

The final part of this exercise is to implement a parser for a small expression language using the parser combinators and type class instances we have implemented so far. Expressions in this language are represented by the following algebraic data type:

```
data Expr = Val Int | Add Expr Expr
```

Given a value of type `Expr`, we can map it to a corresponding `Int` value with the help of the following function (an interpreter):

```
eval :: Expr -> Int
eval (Val n)    = n
eval (Add l r) = eval l + eval r
```

Ex129 With the help of `token` and `ch`, implement the `lparen`, `rparen`, and `plus` parsers. The `lparen` parser should parse `(`, `rparen` should parse `)`, and `plus` should parse `+`. Each may optionally be preceded by whitespace which should be ignored:

```
parse lparen "("     ==> Just ('(', "")
parse lparen "   (" ==> Just ('(', "")
parse rparen ")"     ==> Just (')', "")
parse rparen "   )" ==> Just (')', "")
parse plus "+"       ==> Just ('+', "")
parse plus "   +"    ==> Just ('+', "")
```

Ex130 Implement the `expr` parser which should either accept `val` or `add`.

You should now be able to run `parseAndEval` on strings containing simple arithmetic expressions as shown in the introduction to this lab.

4.18 Effectful Programming

Topics: Monads.

These exercises are about monads. As usual, you can obtain the skeleton code for the exercises by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-effectful-programming
```

For your reference, the definition of the *Monad* type class is given below. A type `m` is a monad if it is an applicative functor and there is a function of type `m a -> (a -> m b) -> m b` which satisfies the monad laws. This function is referred to as "bind" and denoted by the `(>>=)` operator. In Haskell, we can overload the `(>>=)` operator for all types which are monads using a type class:

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure

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

Also for your reference, the monad laws are shown below:

$$\begin{array}{lrcl} \textbf{Left identity} & return\ x \gg= f & = & f\ x \\ \textbf{Right identity} & m \gg= return & = & m \\ \textbf{Associativity} & (m \gg= f) \gg= g & = & m \gg= (\lambda x \to f\ x \gg= g) \end{array}$$

📘 *Recommended reading*: Chapters 12 and 13 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 12 of *Programming in Haskell* (Hutton, 2016).

📘 *Further reading*: the use of monads in functional programming was popularised by *Monads for functional programming* (Wadler, 1995).

Ex131 Suppose that you are given a family tree of the Duck family. The family tree is represented as a Haskell list of pairs which maps the names of members of the Duck family to their respective parent:

```haskell
duckily :: [(String, String)]
duckily =
    [ ("Grandduck", "Grand duckster")
    , ("Baby Duck", "Parent Duck")
    , ("Duckling", "Older duckling")
    , ("Parent Duck", "Grandduck")
    ]
```

Using the `lookup` function and the fact that `Maybe` is a monad, write a function

```
grandduck :: [(String, String)] -> String -> Maybe String
```

which, given a family tree in the format shown above and the name of a member of the family, should retrieve its grandparent if there is one known:

```
grandduck duckily "Baby Duck"   ==> Just "Grandduck"
grandduck duckily "Duckling"    ==> Nothing
grandduck duckily "Parent Duck" ==> Just "Grand duckster"
```

Ex132 Could you define the `grandduck` function if `Maybe` was only an applicative functor? If not, why?

Ex133 The `map` function can be generalised to a `mapM` function which allows the function given as argument to return a monadic computation and sequences those computations:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do
    y  <- f x
    ys <- mapM f xs
    return (y:ys)
```

For example:

```
mapM (safediv 4) [1,2,4] ==> Just [4,2,1]
mapM (safediv 4) [1,0,4] ==> Nothing
```

This version of `mapM` is defined with the help of the `do`-notation, which is just syntactic sugar for `(>>=)` and anonymous functions. Show what the definition of `mapM` given above would look like without the `do`-notation, *i.e.* what the compiler desugars it into.

Ex134 Can you define the same function as `mapM`, but only with an `Applicative` constraint instead of `Monad`?

Ex135 The `zipWith` function we encountered previously can also be generalised to functions which return monadic computations:

```
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

Implement this function so that the following expressions result in the values shown:

```
zipWithM safediv [] [1,2,3]     ==> Just []
zipWithM safediv [4,5,6] []     ==> Just []
zipWithM safediv [4,5] [1,2,0] ==> Just [4,2]
zipWithM safediv [4,5,6] [1,0] ==> Nothing
zipWithM (\x y -> [x,y]) [1,2] [3,4]
==> [[1,2],[1,4],[3,2],[3,4]]
```

The `Either` type is similar to the `Maybe` type in that it can be used to indicate success and failure. Unlike the `Maybe` type, whose `Nothing` constructor takes no arguments, the `Left` constructor of `Either a b` takes one argument of type a which can be used to provide more information about the reason of the failure.

Ex136 The `Either` type is a functor. Complete the instance of `Functor` for it so that:

```
fmap (+5) (Left "Witter") ==> Left "Witter"
fmap (+5) (Left Nothing)  ==> Left Nothing
fmap (+5) (Right 8)       ==> Right 13
```

Ex137 The `Either` type is an applicative functor. Complete the instance of `Functor` for it so that the following expressions evaluate to the values shown:

```
pure (+4) <*> Right 42      ==> Right 46
pure (+4) <*> Left "Koan"   ==> Left "Koan"
(+) <$> Right 4 <*> Right 2 ==> Right 6
(+) <$> Left 4 <*> Right 2  ==> Left 4
(+) <$> Right 4 <*> Left 2  ==> Left 2
```

Ex138 The `Either` type is a monad. Complete the instance of `Monad` for it so that the following expressions evaluate to the values shown:

```
Right 5 >>= \x -> if odd x then Left "Odd!" else Right (x*2)
==> Left "Odd!"
Right 4 >>= \x -> if odd x then Left "Odd!" else Right (x*2)
==> Right 8
Left "!" >>= \x -> if odd x then Left "Odd!" else Right (x*2)
==> Left "!"
```

Ex139 Prove that the monad laws hold for your instance of `Monad` for the `Either` type.

Ex140 The `Parser` type from the previous lab is a monad. Write a `Monad` instance for it. Once completed, you can write parsers which can make choices depending on what has been parsed so far. As a simple example, you could write a parser which parses the same character twice:

```
twice :: Parser String
twice = do
    c <- ch (const True)
    d <- ch (==c)
    return [c,d]
```

Using this parser would then give you the following results:

```
parse twice "aaab" ==> Just ("aa","ab")
parse twice "bbab" ==> Just ("bb","ab")
parse twice "bcaa" ==> Nothing
```

Ex141 (*Mini Project*) If you are feeling particularly adventurous and want to test your skills at using the `Parser` type, try writing a parser for the JSON format[4] using it. Alternatively, you could try using an existing parser library from Hackage, such as `parsec`[5] or `megaparsec`[6] to write a JSON parser. If you ever need a real JSON parser for a project in Haskell, the `aeson`[7] library implements one which is very efficient (at the cost of having bad error messages).

---

[4]`http://www.json.org`
[5]`http://hackage.haskell.org/package/parsec`
[6]`http://hackage.haskell.org/package/megaparsec`
[7]`http://hackage.haskell.org/package/aeson`

4.19 Input & Output

Topics: The `IO` monad.

These exercises are about the `IO` monad. As usual, you can obtain the skeleton code for the exercises by cloning the respective repository from GitHub:

```
$ git clone https://github.com/fpclass/lab-io
```

As we know, Haskell is a purely functional programming language – that is, functions are pure and therefore free of side effects. Input and output (I/O), such as reading from or writing to files, is a side effect. However, I/O is extremely useful and important for writing programs in practice. We have already seen that effects can be encoded in Haskell using monads, allowing us to abstract over behaviours that we want to happen implicitly in our program. Haskell uses the same approach to let us perform I/O by providing the `IO` monad. Now, "hiding" side effects in a monad does not magically make programs that use it pure, but Haskell's type system prevents us from using `IO` inside of pure functions, thus keeping them pure. This effectively means our Haskell programs end up having two layers: an outer layer which is impure and can perform I/O and an inner layer which is pure and cannot perform I/O. The I/O layer can call pure functions though.

Recommended reading: Chapter 9 of *Learn you a Haskell* (Lipovača, 2011) or Chapter 10 of *Programming in Haskell* (Hutton, 2016).

Further reading: the `IO` monad was first described in *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell* (Peyton Jones, 2001).

Ex142 You can find an overview of types and functions related to the `IO` monad that are part of the standard library in the documentation for the `System.IO` module[8]. Your first task is to complete the definition of `extractFirstLine` which is given two file handles (values of type `Handle`) as arguments and should read the first line from the first handle and write it to the second handle. Check that your solution works by running `stack test` as usual.

Ex143 Complete the definition of `replicateFirstLine` which should read the first line from input file and write it to the output file five times.

Ex144 Complete the definition of `reverseFile` which should read all lines from the input file and write the lines in reverse order to the output file.

---

[8]https://hackage.haskell.org/package/base/docs/System-IO.html

Ex145 Complete the definition of `readKeyValuePair` which should parse a `String` value of the form `"key=value"` into a pair of the form `("key", "value")`. You may assume that the input string is always correctly formatted. Note that both keys and values may contain spaces.

Ex146 Complete the definition of `readDictionary` which should read all lines from the input file, each of which is of the form `"key=value"`, and use `readKeyValuePair` to convert them into corresponding key-value pairs.

Ex147 With the help of `readDictionary`, complete the definition of `writeGrandDucks`. This function takes three file paths as arguments: the location of a file containing key-value pairs that maps ducks to their parents, the location of a file containing the names of ducks (one per line), and the location of an output file. Unlike in the previous tasks, you will have to open the files yourself to obtain handles for them. For each line in the second file, try to determine the duck's grandduck using the mappings from the first file: if you can determine the grandduck, write its name to the output file. If you cannot determine the grandduck, write a dash to the output file. The output file should therefore end up with as many lines as the file that contains the input duck names.

Example dictionary file:

```
Duckling=Parent duck
Parent duck=The Ancient Quack
```

Example duck list:

```
Duckling
Parent duck
```

Example output:

```
The Ancient Quack
-
```

4.20 Kinds

Topics: Kinds and data type promotion

Kinds are the "types of types". While types describe the sort of values that an expression may evaluate to and prevent us from writing *e.g.* functions that break at runtime, kinds are useful to prevent us from doing "wrong" things with types themselves. For example, we know that a type constructor such as `Maybe` is parametrised over a type and must therefore be applied to one type as argument before `Maybe` yields a type itself. Kinds formalise this idea and allow us to say that *e.g.* `Maybe` is of kind `* -> *`. The kind `*` represents all types and the kind `* -> *` represents all type constructors which expect a single type as argument before yielding a type themselves, such as `Maybe`. Therefore, *e.g.* `Maybe Bool` is of kind `*`.

The skeleton code for these exercises can be obtained as usual with the following command:

```
$ git clone https://github.com/fpclass/lab-kinds
```

Note that there are no tests for this lab and all exercises can be completed in the REPL which can be opened as usual with `stack repl`. There are some commands that are supported by the REPL which you may find useful for this set of exercises and for the exercises on *Type-level programming*:

`:k TYPE`	Infers the kind of `TYPE`.
`:kind! TYPE`	Reduces `TYPE` to a normal form.

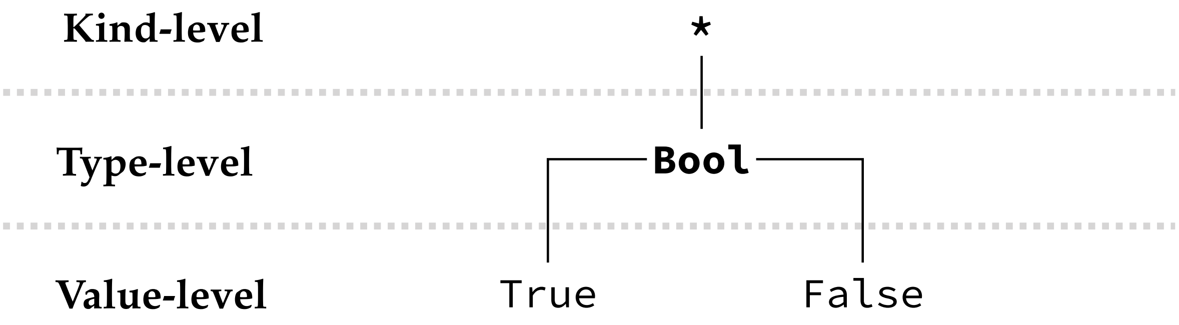
Ex148 Just like you can ask the Haskell compiler to infer the types of expressions for you, you can also ask it to infer the kinds of types for you. Try running the following commands in the REPL to validate that what we described above is correct:

- `:k Maybe`
- `:k Maybe Bool`
- `:k Maybe Int`
- `:k Maybe (Maybe Bool)`
- `:k []`
- `:k [Bool]`

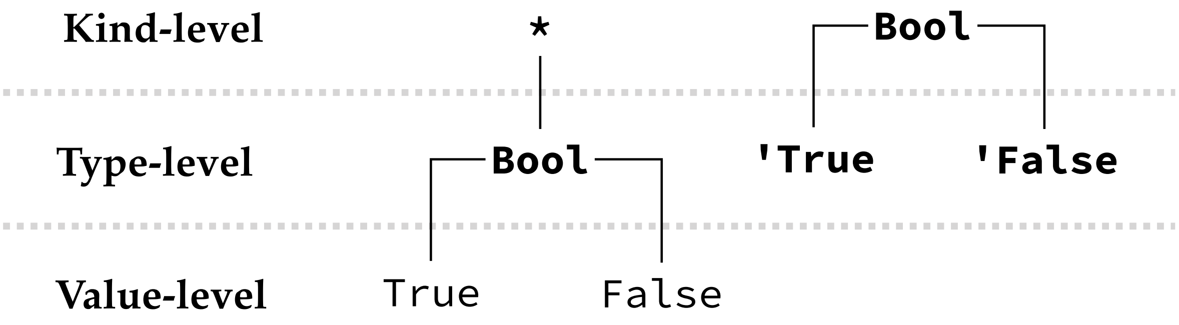
Ex149 What is the kind of the `Compose` type from the exercises about functors?

```haskell
data Compose f g a = Compose (f (g a))
```

Ex150 GHC provides type-level booleans out-of-the-box for us. Ordinarily, we can think of the `Bool` type as follows:



That is, `Bool` is a type and there are two values of this type: `True` and `False`. As an ordinary type, it is of kind $*$ (read as *type*). With `-XDataKinds` (*i.e.* type promotion) enabled, the `Bool` type is automatically promoted to its own kind and its data constructors, `True` and `False`, are automatically promoted to types of that kind, all *in addition to the above*. We can visualise this as:



In addition to the `Bool` type with its `True` and `False` constructors, we now have a kind that is also named `Bool`. There are two types of kind `Bool`: `'True` and `'False`. Run the following commands in the REPL to validate what we have described:

- `:k Bool`
- `:t True`
- `:k 'True`
- `:k True`

Note that the last two commands are the same. Because types and values are different namespaces, we can often just write `True` instead of `'True` to refer to the type named `True`. However, to avoid ambiguity it is always good practice to explicitly include the `'`.

Ex151 GHC also provides type-level lists out-of-the-box for us when `-XDataKinds` is enabled. Try the following in the REPL:

- `:k []`
- `:t (:)`
- `:t []`
- `:k '[]`
- `:k (:)`
- `:k Int : '[]`

4.21 Type-level programming

Topics: Kinds, phantom types, GADTs, singleton types, pattern matching with GADTs, data type promotion, closed and open type families.

This final set of exercises is all about type-level programming. The skeleton code can be obtained as usual with the following command:

```
$ git clone https://github.com/fpclass/lab-tlp
```

📕 *Further reading*: for more examples of type-level programming in action, *Fun with type functions* (Kiselyov et al., 2010) is a good read. The first two chapters of *Giving Haskell a promotion* (Yorgey et al., 2012) give an overview of key type-level programming techniques in Haskell.

Ex152 Define a closed type family `Not` which performs boolean negation at the type-level. Check that your solution works correctly by running `stack test` as usual.

Ex153 Once defined, you should be able to use the REPL commands to infer the types and kinds of your new definition. Try the following:

- `:k Not`
- `:k Not True`
- `:kind! Not True`

Ex154 Modify the definition of `SBool` to define a singleton type for booleans. This type should have kind `Bool -> *` and two constructors named `STrue` and `SFalse` with appropriate types. Once you have defined this type, verify in the REPL that the type has the correct kind and that the constructors have the correct types. The unit tests will also ensure that `STrue` and `SFalse` have the correct types.

Ex155 Having a singleton type for booleans is useful as we can now keep track of the value of a boolean variable at the type-level and therefore at compile-time. This allows us to define functions of types such as:

```
inot :: SBool b -> SBool (Not b)
```

That is, given a value of type `SBool b` where `b` is a type of kind `Bool` corresponding to the value, `inot` should return a boolean whose value is the negation of `b`. Implement this function now so that we get the expected behaviour:

```
inot STrue  ==> SFalse
inot SFalse ==> STrue
```

While this is not very interesting on the term-level, what happens if you ask the REPL for the types of these expressions?

- `:t inot STrue`
- `:t inot SFalse`

Ex156 Could you define similar functions for other boolean operations, such as `and`?

Ex157 Given a type that is known at compile-time, we may wish to convert it to a corresponding value on the value-level. This process is in general known as *reification* and can be accomplished with the help of a suitable type class. We now want to do this for type-level booleans. You are already given the definition of a suitable type class, named `KnownBool`. Implement suitable instances of this type class so that `boolVal` can be used in the following ways:

```
boolVal (Proxy :: Proxy True)  ==> True
boolVal (Proxy :: Proxy False) ==> False
```

The aim of this last part of these exercises is to implement *heterogeneous lists* in Haskell. The "ordinary" lists that we have come across in Haskell are homogeneous: that is, every element has the same type. For example, the following is a valid list in Haskell because all elements have the same type:

```
[4,8,15,16,23,42] :: [Int]
```

However, the following is not a valid list in Haskell because its elements have different types:

```
[True,"Duck"] -- not well typed
```

This is because lists in Haskell need to be parametrised by the element type: the list type constructor `[]` has kind `* -> *`. The definition of lists assumes that every element has that type:

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

In order to implement lists where the elements can have different types, we need to be able to parametrise a list by the types of all of its elements. In other words, we need a type constructor of kind `[*] -> *`. That is, a type constructor which requires a list of types as argument.

Ex158 With the help of type-level lists, complete the definition of `HList` so that it has two constructors: `HNil` which represents an empty, heterogeneous list and `HCons` which adds an element to a heterogeneous list. Some examples of what should work successfully in the REPL once you are done:

```
*Lab> :t HNil
HNil :: HList '[]
*Lab> :t HCons True HNil
HCons True HNil :: HList '[Bool]
*Lab> :t HCons 4 (HCons True HNil)
HCons 4 (HCons True HNil) :: Num a => HList '[a, Bool]
```

Ex159 Implement the `hhead` function, which should work just like `head` does on ordinary lists, but for heterogeneous lists.

Ex160 Define suitable instances of the `Show` type class so that we can use the `show` function on heterogeneous lists. For example:

```
show HNil                          ==> "[]"
show (HCons 4 HNil)                ==> "4 : []"
show (HCons "cake" (HCons 4 HNil)) ==> "\"cake\" : 4 : []"
```

Once implemented, all tests should pass.

Ex161 (*Difficult*) Replace your instances of the `Show` type class that you wrote for the previous exercise with new ones so that the `show` function on heterogeneous lists works as follows instead:

```
show HNil                          ==> "[]"
show (HCons 4 HNil)                ==> "[4]"
show (HCons "cake" (HCons 4 HNil)) ==> "[\"cake\", 4]"
```

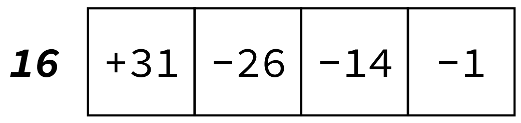
Hint: Constraint kinds[9] may help you solve this task.

[9] `http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/`

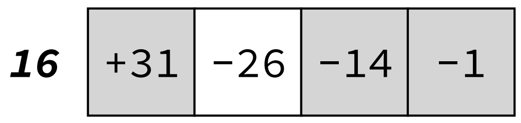
λ.5 Coursework I

5.1 The Large Arithmetic Collider

The goal of this coursework is to implement a Haskell program which can solve a challenging combinatorial game that we have named the *large arithmetic collider*. The game focuses on a grid containing mathematical operations. To explain how it works, let us start with a simple example: consider the following row containing the operations `+31`, `-26`, `-14`, and `-1` from left to right:



You will also note that this row is annotated with the number 16 on the left. The goal now is to determine which of the operations contained in the cells result in that number when applied from left to right, starting with 0. The solution for this example is shown below, where the cells containing operations that can be used to obtain the target number are shaded in grey (we refer to them as "enabled" cells):



As we can easily see, this is a solution because $0 + 31 - 14 - 1 = 16$. Since there are 4 cells in this example and each cell can either be enabled or not, there are $2^4 = 16$ possible configurations to explore when searching for a solution for this grid.

Although there is only one solution for this example, grids may have more than one solution or, indeed, no solution.

You may notice that we are talking about a "grid", but have only shown a single row to illustrate the basic game mechanics. A more realistic grid is shown below:

	53	**-48**	**-21**	**174**
41	+33	-62	+8	+13
26	+23	-48	+14	+74
56	+82	+28	+73	+56
35	+20	-16	-29	+44

As we can see, every row and every column is annotated with a target number. We must find which operations to enable so that every row results in its target number as described in the previous example and that every column also results in its target number. The rules for columns are the same as for rows: we start with 0 and apply each active operation from top to bottom. The solution for this grid is shown below:

	53	**-48**	**-21**	**174**
41	▨ +33	-62	▨ +8	+13
26	+23	▨ -48	+14	▨ +74
56	+82	+28	+73	▨ +56
35	▨ +20	-16	▨ -29	▨ +44

Finding a solution for this grid is more difficult. In this example, there are $4 \times 4 = 16$ cells and the number of possible configurations to explore is therefore $2^{16} = 65535$. In general, grids can have arbitrary dimensions and so the number of possible configurations is $2^{rows \times columns}$. Fortunately, such grids are still fairly simple for a computer to solve.

5.1.1 Rotations

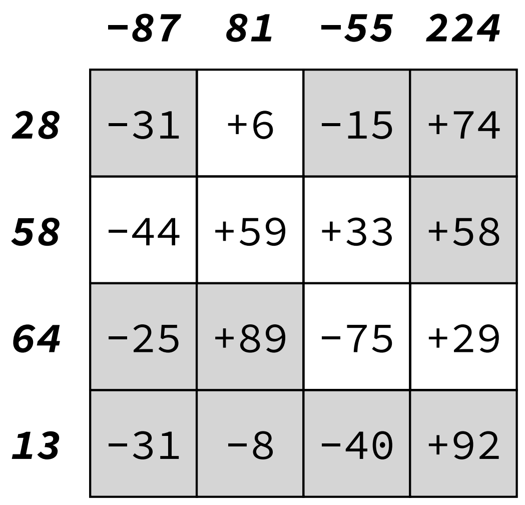
To further increase the difficulty of the game, there is one more game mechanic. While some grids can be solved as they are, we can also consider grids where rows or columns must be *rotated* first. For example, consider the following grid:

	-87	**81**	**-55**	**224**
28	-31	+6	-15	+74
58	-44	+59	+33	+58
64	-25	+89	-75	+29
13	+92	-31	-8	-40

This grid has no solutions as it is. However, we can find solutions if we change the layout of the cells by rotating some of the rows or columns. The rules for this are as follows: we always perform rotations from right to left or bottom to top. Therefore, for a given grid, there are always *rows* + *columns* possible rotations. For the example shown above, we can solve it by rotating the last row of the grid:

	-87	**81**	**-55**	**224**
28	-31	+6	-15	+74
58	-44	+59	+33	+58
64	-25	+89	-75	+29
13	-31	-8	-40	+92

We refer to applying one such rotation as a *move*. If a grid cannot be solved in zero moves (*i.e.* without rotations), then our goal is to try and solve it in *as few moves as possible*. With the last row rotated as shown above, there is now a solution for the resulting grid:



	-87	**81**	**-55**	**224**
28	-31	+6	-15	+74
58	-44	+59	+33	+58
64	-25	+89	-75	+29
13	-31	-8	-40	+92

Finding a solution for a grid that can only be solved with rotations is much harder than finding solutions for one that can be solved in zero moves. In theory, we can use rotations to create an arbitrary arrangement of cells in the grid, thus leaving us with $(rows \times columns)!$ many arrangements of cells. That means that for a 4×4 sized grid such as the one shown above, there are $20,922,789,888,000$ many arrangements of cells. Each cell can then either be enabled or not, so for each of those arrangements there are $2^{4\times4}$ possible solutions, leaving us with a total state space of $(4 \times 4)! \times 2^{4\times4}$ many possible solutions for a 4×4 grid. However, fortunately we do not need to search this entire state space since just finding a solution in itself does not tell us how to get there from the grid we start with, so a smarter approach is needed to find a solution in as few moves as possible!

5.2 Before you get started

Before you get started with this coursework, we *strongly recommend* that you complete all lab exercises up to and including those on data types (Section 4.11). You will find them all extremely helpful in preparing you for the tasks in this coursework!

If you need help while working on the coursework, remember that help is available from me (Michael) and the teaching assistants. I am usually pretty quick to reply to questions, particularly on the Slack. Additionally, see Section 2.6 for other resources you may find useful.

5.3 Getting started

In order to get started with the coursework, you need to get hold of the skeleton code and ensure that it compiles successfully on your machine.

5.3.1 Obtaining the skeleton code

There are three different ways in which you can obtain the skeleton code for this coursework. All of them are explained below alongside their various advantages and disadvantages:

ℹ We encourage you to use version control for your work and the first two options below assume you have basic familiarity with version control using `git`. You can read Section 2.4 for a `git` crash course or `https://git-scm.com/book/en/v2` for a more comprehensive introduction.

Option A: Private fork By following the GitHub Classroom link below, you can create a private fork of our git repository with the skeleton code. This requires a GitHub account, but has the advantage that you have your own private copy of our repository on GitHub that you can write to. That would allow then you to work easily share your work between machines in the labs and at home:

`https://classroom.github.com/a/eQTilEdh`

Once you have accepted the assignment, you can then clone your fork of the skeleton code to your machine with the usual `git clone` command where `[username]` is your GitHub username:

```
$ git clone https://github.com/fpclass/2021-cswk1-[username]
```

Option B: Clone If you do not wish to create a GitHub account or host a copy of your repository there, then you could instead just clone our repository with:

```
$ git clone https://github.com/fpclass/large-arithmetic-collider
```

You will be able to `git commit` changes to your local copy of the repository, but you will not be able to `git push` them. This is sufficient if you are only planning to work on the coursework from one place (*e.g.* only the lab machines but not your personal computer).

Option C: Archive If GitHub should be unavailable or you do not have `git` installed your machine, you can download a `.zip` file with the skeleton code from the module website.

5.3.2 Working with the skeleton code

You may wish to verify that the code compiles and that all tests fail by entering the `large-arithmetic-collider` directory and running `stack test`:

```
$ cd large-arithmetic-collider
$ stack test
```

Running `stack test` will compile your code, run a bunch of unit and property tests on it, and give you a rough indication of how complete your solution is: the more tests pass, the more complete it is. Initially, not all tests will be run (those that are not run will show “SKIP” as their status) since they depend on working implementations of other functions. As you make progress with the coursework and complete functions so that they pass their tests, the other tests will then be executed whenever you run `stack test`.

You can also use `stack build` to just compile your code and then `stack exec collider` to run the game (or just `stack run` to do both). Note that the game will of course not work properly until you have implemented the tasks for this coursework. Specifically, the first two campaigns will not work until you have implemented the functions up to and including `solve` and the last two campaigns will not work until you have implemented the remaining functions, including `steps`. You can run `stack repl` to load up the REPL, which is useful for debugging.

The skeleton code contains a bunch of files, most of which you do not need to touch. The most important file is `src/Game.hs` which contains the definitions you will need to complete in order to implement the game. There are some definitions to get you started. Firstly, the arithmetic operations that may be contained in cells of the grids are represented as an algebraic data type where each constructor represents one type of operation along with its operand:

```haskell
data Action
  = Add Int
  | Sub Int
```

Cells themselves are also represented as an algebraic data type comprised of a boolean value indicating whether the cell is enabled or not and an `Action` value representing the arithmetic operation contained in the cell:

```haskell
data Cell = MkCell Bool Action
```

Rows are comprised of a target number for the row and a list of cells:

```
data Row = MkRow Int [Cell]
```

Grids are comprised of a list of target numbers for the columns and a list of all the rows in the grid:

```
data Grid = MkGrid [Int] [Row]
```

Finally, we have a definition for a little helper type to represent directions:

```
data Direction = L | R
```

5.4 Task

Complete all definitions in `src/Game.hs` so that the game works as described above. The following function stubs in `src/Game.hs` need to be implemented:

1. `eval :: Action -> Int -> Int`
   This function should apply an arithmetic operation represented by an `Action` value to an accumulator and return the result. For example, `eval (Add 5) 10` should evaluate to `15`.

2. `apply :: Cell -> Int -> Int`
   This function should apply the arithmetic operation contained in a `Cell` value to an accumulator and return the result if the cell is enabled. For example, `apply (MkCell True (Add 5)) 10` should evaluate to a result of `15` while `apply (MkCell False (Add 5)) 10` should evaluate to `10`.

3. `result :: [Cell] -> Int`
   This function should determine the result of evaluating all the enabled arithmetic operations in a list of cells, starting with 0. For example, evaluating `result [MkCell True (Add 3), MkCell False (Add 5)]` should result in `3`.

4. `states :: Cell -> [Cell]`
   A function which returns a list with *exactly* two elements that represent the two different states a cell can be in. For example:

   ```
       states (MkCell False (Add 5))
   ==> [MkCell True (Add 5), MkCell False (Add 5)]
   ```

5. `candidates :: [Cell] -> [[Cell]]`
   A function which, given a list of cells in a row, produces all possible combinations of states for those cells. For example:

```
    candidates [MkCell False (Add 5), MkCell False (Sub 1)]
==> [ [MkCell False (Add 5), MkCell False (Sub 1)]
    , [MkCell False (Add 5), MkCell True (Sub 1)]
    , [MkCell True (Add 5), MkCell False (Sub 1)]
    , [MkCell True (Add 5), MkCell True (Sub 1)]
    ]
```

6. `solveRow :: Row -> [Row]`
   This function should find all solutions for a given row. For example:

```
    solveRow (MkRow 4 [ MkCell False (Add 4)
                      , MkCell False (Sub 7)
                      ])
==> [MkRow 4 [MkCell True (Add 4), MkCell False (Sub 7)]]
```

7. `solve :: Grid -> [Grid]`
   This function should find all solutions for a given grid, without rotating any rows or columns. The cells in the input grid may be in any state. If there are no solutions, an empty list should be returned. For example:

```
solve (MkGrid [2,4] [ MkRow 3 [ MkCell False (Add 2)
                              , MkCell False (Add 1)
                              ]
                    , MkRow 3 [ MkCell False (Add 2)
                              , MkCell False (Add 3)
                              ]
                    ])
==> [MkGrid [2,4] [ MkRow 3 [ MkCell True (Add 2)
                            , MkCell True (Add 1)
                            ]
                  , MkRow 3 [ MkCell False (Add 2)
                            , MkCell True (Add 3)
                            ]
                  ]]
```

8. `rotate :: Direction -> [a] -> [a]`
   A function which rotates the items in a list to the left or right depending on the specified direction. For example, `rotate L [1,2,3]` should evaluate to `[2,3,1]`. Note that although gameplay will always use rotations to the left (*i.e.* `rotate L`), `rotate R` is required for the tests to work correctly.

9. `rotations :: Grid -> [Grid]`
   Given a grid, this function should return a list of grids containing all possible

ways to rotate the input grid by one rotation. This means the resulting list should normally have *rows* + *columns* many elements.

10. `steps :: Grid -> [Grid]`
    If a grid cannot be solved without rotating it, this function should return a list of grids representing the shortest sequence of rotations which lead to a solution. I.e. the list returned by `steps inputGrid` should contain as many elements as rotations are required to reach a solution starting from `inputGrid`. Note that `inputGrid` should not be included in the resulting list. The last grid in the list returned should be the solution (with the right cells enabled) and each grid should differ from the previous grid in the list by exactly one rotation. You may assume that all grids given to `steps` as arguments will require at least one move.

5.5 Remember...

There are some important things you should keep in mind when working on the tasks for this coursework:

- You probably want to implement the functions in the order in which they are shown. Functions you define are designed to be used in the implementations of some later functions. As examples, `solve` is probably useful for the definition of `steps` and `eval` for the definition of `apply`.

- You may find that some of the above functions are significantly more complicated to implement than others. If you find this to be the case, think about how you could break it down into smaller, simpler functions which solve smaller parts of the problem. You are allowed (and encouraged) to define your own functions in addition to those that you are required to implement.

- You may import and use libraries. This includes anything from the `base` package[1], such as the `Data.List` module. You can even add other packages, such as packages which implement various data structures, to the dependencies in `package.yaml` if you like, provided that they do not solve significant parts of the coursework for you. If you are in doubt about whether a particular package is allowed, feel free to ask. If a package you want is not yet installed on the DCS machines, let me (Michael) know and I can install it for you.

- You should not change the types of the functions that you are required to implement since that would break the tests. If you want to do this for an

[1]`https://hackage.haskell.org/package/base`

extension you have planned, make a copy of your code and modify that. Then submit two folders: one with the version of your code with the extensions (where the tests may not work) and one without (which passes the tests).

5.6 Solving simple grids

To solve simple grids (*i.e.* those that can be solved in zero moves) you need to implement all functions up to and including `solve`. The functions leading up to `solve` are fairly rigidly structured, but you may find `solve` to be a slight jump in difficulty because you need to decide how you want it to work. A brute-force solution for `solve` could simply enumerate all possible states the grid can be in and then check for which of those all rows and columns result in their targets. You might find this to be slow and inefficient. Think about how you might leverage functions you implemented for the previous parts, such as `solveRow`, to generate fewer grids that need to be checked.

5.7 Solving complex grids

For complex grids (*i.e.* those that require rotations to be solved), you want to *think* of the problem as a tree where the root of the tree is the initial grid. The rotations that can be applied to this initial grid then result in the children of that node and the rotations that can be applied to each of those result in their children and so on. You need to be careful though: you do not want to generate the entire tree since it would be very large and take a long time to generate. Furthermore, some rotations may lead you to end up with grids you already encountered earlier on in the tree. Think about what order you would want to generate and explore such a tree in considering that you are interested in finding a solution in the fewest moves possible.

5.8 Originality & academic practice

This coursework is an individual assignment and the work you submit must be entirely your own work. Students are expected to be familiar with the departmental Student Handbook as well as applicable university regulations. The "Cheating and Plagiarism" section on the handbook page about coursework is particularly relevant:

`https://warwick.ac.uk/fac/sci/dcs/teaching/handbook/coursework/`

Examples of what is not acceptable in the context of this assignment include, but are not limited to, the following:

- Collaborating with others, for example by sharing code, looking at other people's code, or discussing implementation details such as which functions you used to implement a particular definition.
- Copying or adapting code from web sources such as Stack Overflow, GitHub, etc. without attribution. This includes taking code written in other programming languages and translating it to Haskell. You may do this if you include a correct attribution to the source in e.g. a comment in your file, but note that you can only be awarded marks for work you have done yourself.

5.9 Marking & submission

This coursework is worth 15% of the overall module mark. It will be marked out of 100% as follows:

- 40% for *correctness*. You gain full marks here if all parts of the coursework have been completed and are correct according to their specifications. You may use `stack test` as a rough indication for whether your implementation is correct, but there are some issues the tests may not cover, so you should convince yourself manually that everything is implemented as described as well.
- 20% for *documented understanding*. You gain full marks here if all parts of the coursework are commented sufficiently well so that someone who is unfamiliar with your code can understand it. Note that you will be marked on the correctness of what you write: quality matters more than quantity.
- 20% for *elegance*. Definitions should be concise and readable, new functions should be introduced where needed, existing library functions used when applicable, etc.
- 10% for *performance and efficiency*. To do well here, you need to use sensible data structures and your functions should perform as little redundant computation as possible. In your comments, you must also discuss what you have done to test your solution's performance and what you have tried to improve it.
- 10% for *improvements and extensions*. This is an opportunity for you to demonstrate creativity and advanced understanding. You could achieve this in many different ways, such as adding additional tests, functionality, improved algorithms, etc. You may wish to modify `app/Main.hs` as well as other source files or even add new ones. You could also prove some properties about your game on paper. The amount of marks awarded will depend on the complexity and creativity of your extension(s) and improvement(s).

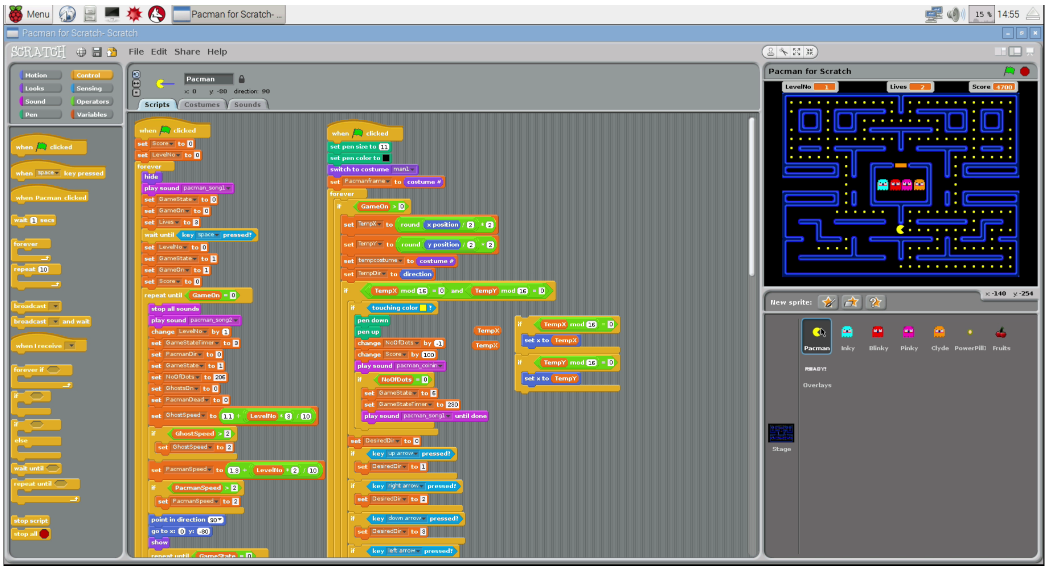
Submit a `.zip` or `.tar.gz` archive of the whole, completed project (not just `Game.hs`) through Tabula by noon on 16 February 2021:

```
https://tabula.warwick.ac.uk/coursework/submission/
        43213a10-18dc-4e87-8364-c648097af402
```

λ.6
Coursework II

6.1 Scratch clone

Scratch[1] is a visual programming language designed to teach programming to children in a fun and graphical way. Programs in Scratch are built by arranging blocks that correspond to different syntactic constructs and connecting them like puzzle pieces. The tool is free to use so you can give it a go if you want! To give you an idea of what it looks like, here is a screenshot of Pac-Man built in Scratch running on a Raspberry Pi:



The goal of this coursework is to implement a simple clone of Scratch. Our clone will consist of two components:

[1] `https://scratch.mit.edu/`

1. A web-based interface which allows users to construct simple programs visually. This is written in JavaScript and is already implemented for you.

2. A Haskell program which handles the evaluation of such programs. This is partially implemented and you will have to finish it.

Your task is to complete the part of the Haskell program responsible for evaluating programs – in other words, you have to write an *interpreter*.

An interpreter is a program which, given some representation of a program as argument, evaluates it. To illustrate this idea, a Haskell implementation of an interpreter for a simple expression language is shown below:

```haskell
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val n)   = n
eval (Add l r) = eval l + eval r
```

Expressions in the language represented by `Expr` consist of the addition operator and integer values. The `eval` function is the interpreter for this language, which determines the value of a given expression.

6.2 Getting started

In order to get started with the coursework, you need to get hold of the skeleton code and ensure that it compiles successfully.

6.2.1 Obtaining the skeleton code

There are three different ways in which you can obtain the skeleton code for this coursework, which are all explained below alongside their advantages and disadvantages:

Option A: Private fork By following the GitHub Classroom link below, you can create a private fork of our git repository with the skeleton code. This requires a GitHub account, but has the advantage that you have your own private copy of our repository on GitHub that you can write to. That would allow then you to work easily share your work between machines in the labs and at home:

`https://classroom.github.com/a/aW9z1Yip`

Once you have accepted the assignment, you can then clone your fork of the skeleton code to your machine with the usual `git clone` command where `[username]` is your GitHub username:

```
$ git clone https://github.com/fpclass/2021-cswk2-[username]
```

Option B: Clone If you do not wish to create a GitHub account or host a copy of your repository there, then you could instead just clone our repository with:

```
$ git clone https://github.com/fpclass/scratch-clone
```

You will be able to `git commit` changes to your local copy of the repository, but you will not be able to `git push` them. This is sufficient if you are only planning to work on the coursework from one place (*e.g.* only the lab machines but not your personal computer).

Option C: Archive If GitHub should be unavailable or you do not have `git` installed your machine, you can download a `.zip` file with the skeleton code from the module website.

6.2.2 Working with the skeleton code

The code should compile out of the box. You can test this by running:

```
$ stack build
```

To start the program, you should run the following command which should result in (roughly) the following output:

```
$ stack run
Starting web server...
Started on http://localhost:52089
Press enter to quit.
```

Note that the port on which the server runs (in this case 52089) is randomised. In order to view the user interface, open your web browser and navigate to the following address, making sure to replace 52089 with whatever port number was shown in the output above:

`http://localhost:52089/`

You can drag together programs using building blocks from the toolbox on the left. However, if you click "Evaluate" at the top right corner of the screen, you will get an error since the interpreter is not yet implemented.

The skeleton code contains a bunch of files, most of which you do not need to touch initially. The most important file is `src/Interpreter.hs` which contains the definitions you will need to complete to get the interpreter to work. There are some definitions to get you started. A program's initial memory is represented as a list of pairs. Each pair represents one variable, consisting of a name of type `String` and a value of type `Int`:

```
type Memory = [(String, Int)]
```

It is possible for things to go wrong when interpreting a program. There are two sorts of errors which may occur. These are represented by the following data type:

```
data Err = DivByZeroError | UninitialisedMemory String
```

The types representing the language itself are defined in `src/Language.hs`. You should have a look at this file yourself, but an overview of the most important types is below. A program is a list of statements:

```
type Program = [Stmt]
```

There are three different forms of statements:

```
data Stmt = AssignStmt String Expr
          | IfStmt Expr [Stmt] [(Expr,[Stmt])] [Stmt]
          | RepeatStmt Expr [Stmt]
```

Assignments, represented by the `AssignStmt` constructor, consists of the name of the variable that we are assigning a value to and the expression whose value we should assign to the variable.

If statements, represented by the `IfStmt`, are more complicated. The first expression is the condition of the "if" clause. The list of statements which follows is the code that should be run if the condition is true. The list of pairs of expressions and lists of statements represent "if else" clauses. Finally, the last list of statements represents the "else" clause.

Repeat statements, represented by the `RepeatStmt` constructor, consist of an expression which determines how many times the repeat loop should be executed and a list of statements which represent the body of the repeat statement.

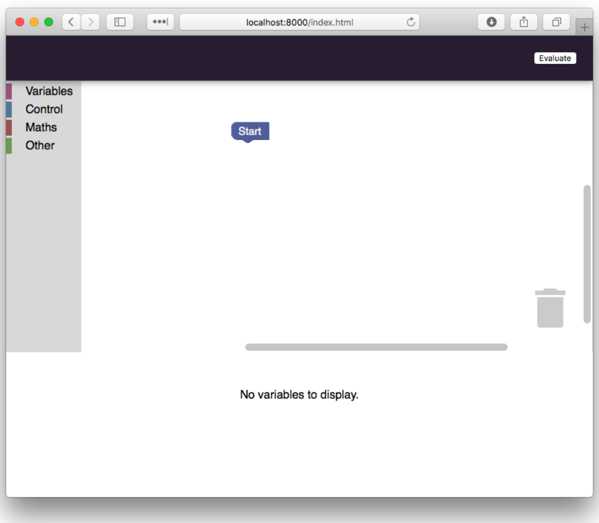
There are also three forms of expressions:

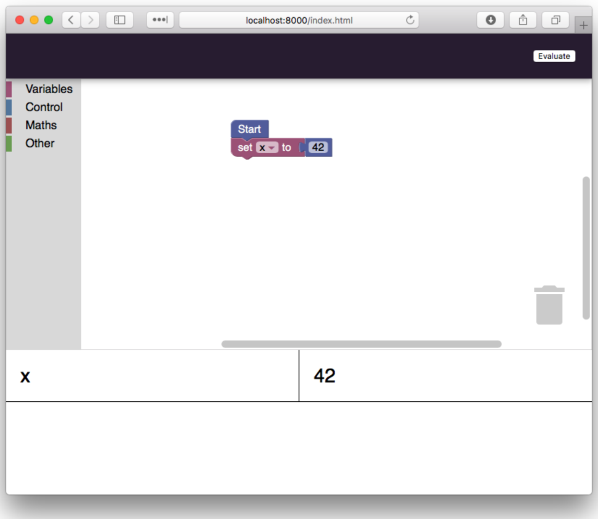
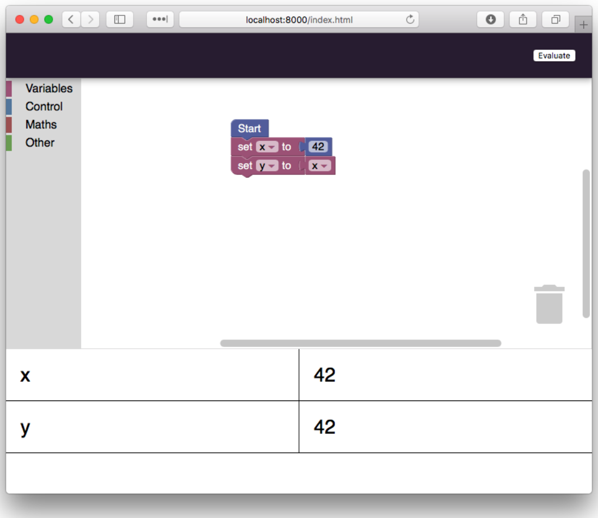
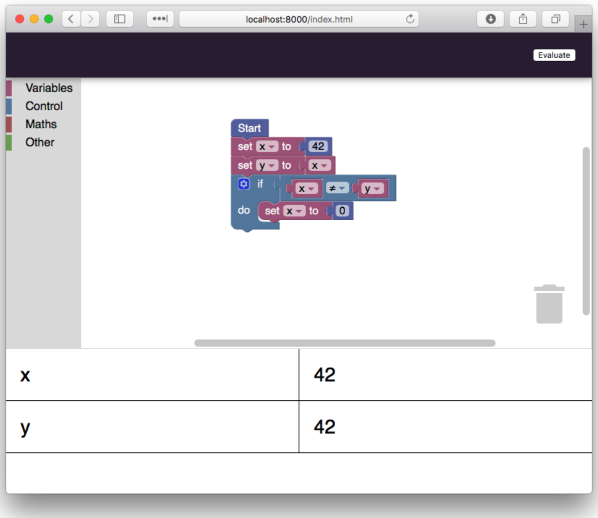
```haskell
data Expr = ValE Int
          | VarE String
          | BinOpE Op Expr Expr
```

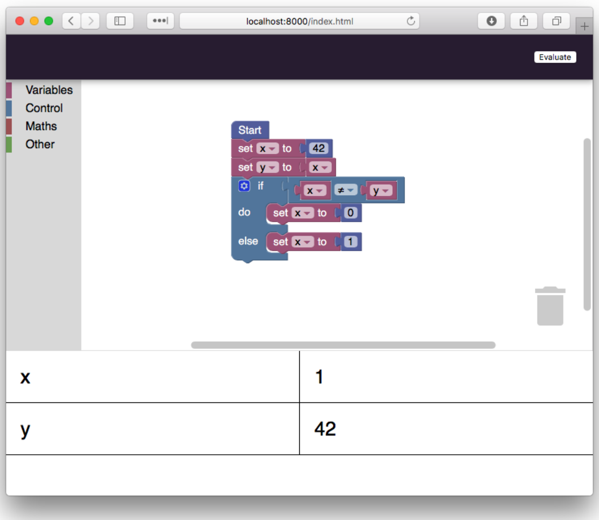
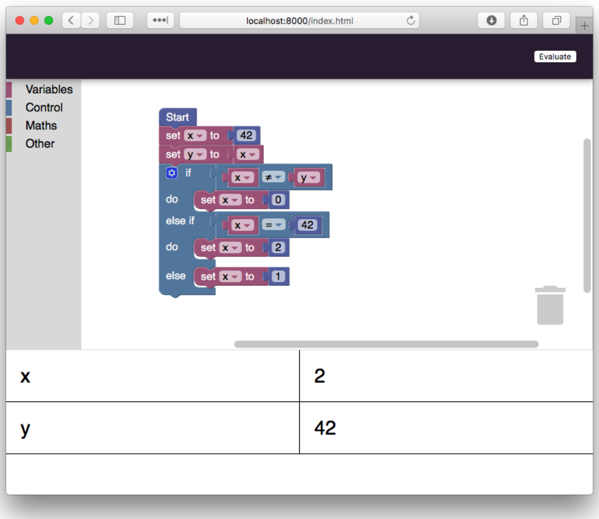
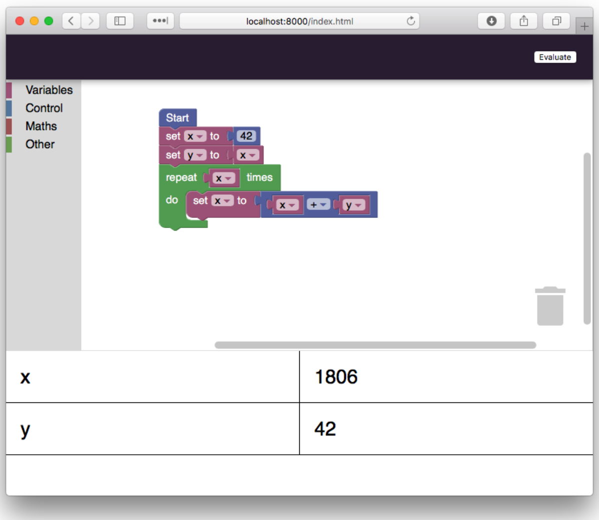
The `ValE` constructor represents integer values, the `VarE` constructor represents variables, and the `BinOpE` constructor generalises binary operators. The `Op` data type in `src/Language.hs` enumerates all available operators.

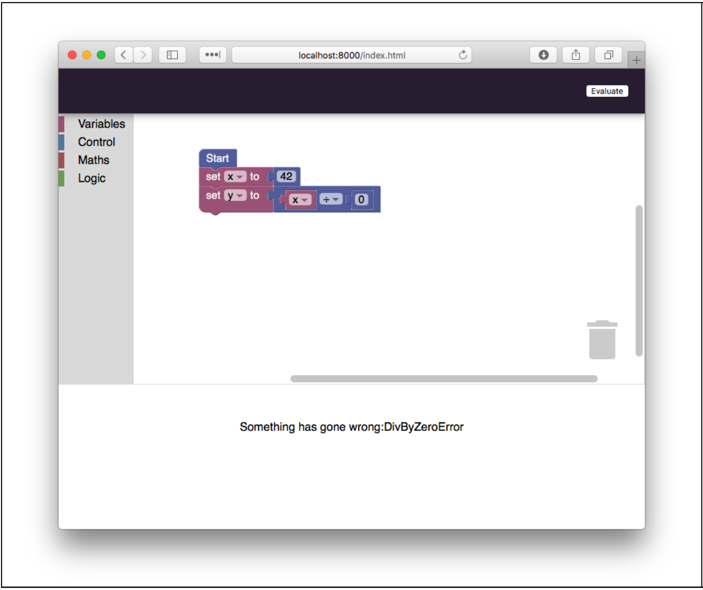
6.3 Task

Complete the definition of the `interpret` function in `src/Interpreter.hs` so that all values of type `Program` can be evaluated correctly according to the rules described below. Programs are sequences of statements and should be evaluated in the order in which they are given. We illustrate all rules for the language with screenshots of the GUI and the expected results:

 localhost:8000/index.html Evaluate Variables Control Maths Other Start No variables to display.	If the program is empty as shown in the screenshot, the initial contents of the memory should be returned.

 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 x 42	Assignment statements should update the memory to the value of their expression.
 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 set y to x x 42 y 42	If a variable occurs in an expression, the corresponding value should be loaded from memory.
 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 set y to x if x ≠ y do set x to 0 x 42 y 42	If the condition of an if statement is true (*i.e.* any non-zero value), then the body of the if clause should be executed.

 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 set y to x if x ≠ y do set x to 0 else set x to 1 x | 1 y | 42	If the condition of an if statement is false (*i.e.* it evaluates to zero), then the body of the else clause should be executed.
 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 set y to x if x ≠ y do set x to 0 else if x = 42 do set x to 2 else set x to 1 x | 2 y | 42	If there are if else clauses present, their conditions should be checked in order after that of the main if clause. If one of them is true, then the corresponding body should be executed.
 localhost:8000/index.html Evaluate Variables Control Maths Other Start set x to 42 set y to x repeat x times do set x to x + y x | 1806 y | 42	Repeat statements evaluate an expression to determine how many times they should run. The body of the repeat statement is then executed that many times.

 localhost:8000/index.html Evaluate Variables Control Maths Logic Start set x to 42 set y to x ÷ 0 Something has gone wrong:DivByZeroError	If a division by zero is attempted, the corresponding error should be returned. *I.e.* `Left DivByZeroError`

There are some details to look out for:

- Internally, logic operators should evaluate to 0 if false or a non-zero value if true. All numeric values other than 0 should be treated as true.
- If an attempt is made to read from a variable which is not in the memory, then the corresponding error should be returned.
- Expressions can be nested arbitrarily deep and expressions of arbitrary complexity may appear in any place where expressions are expected.
- Errors can arise almost anywhere and should be propagated properly.

Running `stack test` will give you a rough indication of how complete your solution is. Running `stack bench` will benchmark your code.

6.4 Originality & academic practice

This coursework is an individual assignment and the work you submit must be entirely your own work. Students are expected to be familiar with the departmental Student Handbook as well as applicable university regulations. The "Cheating and Plagiarism" section on the handbook page about coursework is particularly relevant:

`https://warwick.ac.uk/fac/sci/dcs/teaching/handbook/coursework/`

Examples of what is not acceptable in the context of this assignment include, but are not limited to, the following:

- Collaborating with others, for example by sharing code, looking at other people's code, or discussing implementation details such as which functions you used to implement a particular definition.
- Copying or adapting code from web sources such as Stack Overflow, GitHub, etc. without attribution. This includes taking code written in other programming languages and translating it to Haskell. You may do this if you include a correct attribution to the source in e.g. a comment in your file, but note that you can only be awarded marks for work you have done yourself.

6.5 Marking & submission

This coursework is worth 25% of the overall module mark. It will be marked out of 100% as follows:

- 20% for *correctness*. You gain full marks here if all parts of the coursework have been attempted and are correct. You may use `stack test` as a rough indication for whether this is the case, but there are some things the unit tests do not test for, so you should construct programs in the scratch clone and ensure that everything works as described.
- 20% for *documented understanding*. You should document your code with comments and explain how it works. You gain full marks if all code is documented and explained sufficiently well so that someone who is unfamiliar with your code can understand it.
- 20% for *elegance*. Definitions should be concise and readable, new functions should be introduced where needed, existing library functions used when applicable, monads used where possible, etc.
- 20% for *performance and efficiency*. To do well here, you need to use sensible data structures and your functions should perform as little redundant computation as possible. In your comments, you must also discuss what you have done to test your solution's performance and what you have tried to improve it. You can test performance by running `stack bench` on different versions of your code to see how they compare.
- 20% for *improvements and extensions*. This is an opportunity for you to demonstrate creativity and advanced understanding. You could achieve this in many

different ways, such as adding additional unit tests, functionality, improved algorithms, etc. You may wish to modify `exe/Main.hs` as well as other source files or even add new ones. You could also prove some properties about your interpreter on paper. The amount of marks awarded will depend on the complexity and creativity of your extension(s) and improvement(s).

Submit a `.zip` or `.tar.gz` archive of the whole, completed project (not just `Interpreter.hs`) through Tabula by noon on 23 March 2021:

```
https://tabula.warwick.ac.uk/coursework/submission/
        ac09b7de-5e75-45f9-87e9-ab847d96a001
```

λ.7 Revision project

For revision purposes, you may find yourself wishing to tackle a larger Haskell project than those presented by the lab exercises. For this purpose, we have included a past coursework specification in this chapter which you could work on when revising for the exam. This is entirely optional and no marks are available for completion of this project, but it is fully equipped with tests and benchmarks just like the two current coursework projects.

7.1 Mastermind

The aim of this revision project is to implement the board game *Mastermind* in Haskell with the help of some skeleton code. The game is played by exactly two players: a *codemaker* and a *codebreaker*. At the start of the game, the codemaker makes up a code consisting of four coloured pegs. Pegs are also referred to as symbols. For example:

Yellow, Green, Green, Blue

Each colour (symbol) may be used any number of times in the code, as long as the code has no more than four pegs. There are six colours to choose from. The code is *not* disclosed to the codebreaker, whose objective it is to figure out what the code is. The codebreaker does this by repeatedly *guessing* what the code might be. For example, to start the codebreaker might guess the following code at random:

Green, Red, Blue, Blue

The codemaker then scores the guess according to the following rules:

- For each peg that is in the correct position and has the right colour, the codebreaker scores one coloured marker.

- For each peg that is the right colour but in an incorrect position, the codebreaker scores one white marker.

For example, for the above guess, the codebreaker would score one white marker for the green peg that is in the wrong position and one coloured marker for the blue peg that is in the right position. The codebreaker does *not* score a white marker for the second blue peg. In other words, at most one point is awarded for each peg in the code. The codebreaker then has to use this score to come up with a new guess for the code, which is then scored again, and so on. Once the codebreaker scores four coloured markers, the game is over and the two players switch roles.

7.1.1 Getting started

In order to get started with the revision project, you need to get hold of the skeleton code and ensure that it compiles successfully.

Obtaining the skeleton code

To obtain the skeleton code, clone it from GitHub using the following command:

```
$ git clone https://github.com/fpclass/mastermind
```

You will be able to `git commit` changes to your local copy of the repository, but you will not be able to `git push` them. This is sufficient if you are only planning to work on the coursework from one place (*e.g.* only the lab machines but not your personal computer).

If you wish to fork the repository on GitHub and then clone your fork, feel free to do so.

Working with the skeleton code

You may wish to verify that the code compiles and that all tests fail by entering the `mastermind` directory that was created and running `stack test`:

```
$ cd mastermind
$ stack test
```

Running `stack test` will compile your code, run a bunch of unit tests on it, and give you a rough indication of how complete your solution is (the more tests pass, the more complete it is). Running `stack bench` will run a set of benchmarks on your code. You can also use `stack build` to just compile your code and then `stack exec mastermind` to run the program. Alternatively, you can run `stack repl` to load up the REPL, which is useful for debugging.

The skeleton code contains a bunch of files, most of which you do not need to touch. The most important file is `src/Game.hs` which contains the definitions you will need to complete in order to implement the game. There are some definitions to get you started. Firstly, the number of pegs per code is defined as:

```haskell
pegs :: Int
pegs = 4
```

Ideally, your solution should still work even if this number is modified. We represent colours using characters from `a` to `f` and refer to them as symbols:

```haskell
type Symbol = Char

symbols :: [Symbol]
symbols = ['a'..'f']
```

Again, your solution should continue to work even if you modify how many symbols there are and which characters are used to represent them. A code is a list of symbols:

```haskell
type Code = [Symbol]
```

Codes are scored using coloured and white markers. We define scores to be pairs of integers where the first component of the pair represents the number of coloured markers and the second component represents the number of white markers:

```haskell
type Score = (Int, Int)
```

A player is either human or a computer:

```haskell
data Player = Human | Computer
```

The initial codemaker is defined as a constant:

```haskell
codemaker :: Player
codemaker = Human
```

You can change this value to determine who goes first. Finally, the computer's first guess is defined as:

```haskell
firstGuess :: Code
firstGuess = "aabb"
```

You can change this value to change the computer's first guess, but note that values other than `"aabb"` may cause the computer to take more guesses to crack the code.

7.1.2 Five-guess algorithm

Donald Knuth described an algorithm for Mastermind which, for every code with four pegs, takes a computer no more than five guesses to solve. The algorithm works as follows:

1. Let S be a set of all possible codes (`"aaaa"`, `"aaab"`, ..., `"ffff"`).
2. Let the first guess be `"aabb"`.
3. Get the codemaker to score your guess.
4. If the score has four coloured markers, then the guess was correct.
5. Otherwise, remove all codes from S which would result in a different score. In other words, we know that the code is somewhere in S, so it can only be one which results in the same score for the guess as the one we got from the codemaker.
6. Find the next guess as follows. If there is only one code left in S, use it. Otherwise, for every possible code c (not just those left in S):
   (a) For each possible score s ($(0,1)$, $(0,2)$, ..., $(4,0)$):
      i. Determine how many other codes would be eliminated from S. That is, if the next guess were c and it would get a score of s, how many codes would that eliminate from S – i.e. how many codes with different scores would there be?

   Choose the code which is guaranteed to eliminate the most options from S. This is calculated in the above step by calculating the minimum of eliminations for each code across all the possible scores it might get. In the case of multiple codes producing the same number of guaranteed eliminations, a code which is still a member of S should be picked over one which is not.
7. Go to Step 3.

7.1.3 Task

Complete all definitions in `src/Game.hs` so that the game works as described above and that the computer never takes more than five guesses to figure out a code. The following function stubs in `src/Game.hs` need to be implemented:

1. `correctGuess :: Score -> Bool`
   This function should determine whether a `Score` value represents a winning guess – *i.e.* one where the number of coloured markers matches `pegs` and there are no white markers.

2. `validateCode :: Code -> Bool`
   This function should determine whether a given `Code` value is valid: the code should contain `pegs`-many symbols and all the symbols should be elements of `symbols`.

3. `codes :: [Code]`
   This list should contain all possible codes of length `pegs` using elements from `symbols`. There should be no duplicates.

4. `results :: [Score]`
   This list should contains all possible scores for codes of length `pegs`. There should be no duplicates.

5. `score :: Code -> Code -> Score`
   This function should score a code according to the rules described above. This function should be commutative, so that it does not matter whether the code or the guess is given as first argument and vice-versa.

6. `nextGuess :: [Code] -> Code`
   This function should determine the next guess, given the current S represented as a list of codes.

7. `eliminate :: Score -> Code -> [Code] -> [Code]`
   This function should eliminate all codes from a given S, represented as a list of codes, with the help of the most recent guess (the `Code` argument) and the score which was obtained for it from the codemaker (the `Score` argument).

λ.8 SOLUTIONS

This section contains model answers for selected exercises along with descriptions of how they could be derived. You should only read these for revision purposes or once you have completed the exercises yourself. Remember that it is better for you to ask for help in figuring something out than to just jump to the solutions.

8.1 Recursive functions

Ex41 *Solution*: We know that the goal of `elem` is to determine whether some value of some type `a` is contained in a list where the elements are of type `a`. We also know that we are supposed to use explicit recursion to solve the problem.

Whenever we use recursion, we need to think about what the simplest possible case(s) are that a function might want to solve. In the case of `elem`, we know that the thing we are looking for in the list, let's call it `x`, won't change – we only want to search through the list. Therefore, the simplest case we need to consider is the one where we are looking for `x` in the empty list. Of course `x` will not be in the empty list, so that case always trivially evaluates to `False`. The corresponding definition in Haskell is now very straight-forward:

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
```

Now that we have an implementation for the simplest case, we need to solve the recursive cases. As we have determined above, we know that the element we are looking for won't change, so we can start by writing the following for the next equation (this will not compile yet):

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x
```

For the second argument, we know it can't be the empty list since we already have an equation which covers that case. Consequently, we are left with the non-empty list. We use pattern matching to break the non-empty list into its head, which we name `y`, and its tail, which we name `ys` (this still won't compile yet):

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) =
```

To come up with the right-hand side of the equation, we assume that we already have a working implementation of the `elem` function. That means, we can solve the problem for `ys`:

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = elem x ys
```

Now we are nearly there! With the current definition, `elem` just goes through the whole list given as argument and eventually returns `False` when it reaches the empty list. So now we need to check whether some element from the list `y` is equal to `x`. If so, we can return `True` and otherwise, we should keep looking. One possible way in which we could write this is using guards:

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys)
  | x == y    = True
  | otherwise = elem x ys
```

This implementation now does what we want. Also recall that guards are just syntactic sugar for (nested) if expressions, so this definition of `elem` is equivalent to the following:

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = if x==y then True else elem x ys
```

This version is not as nice as the one with guards, though. In general, using guards is nicer than using if expressions. However, there is an even more elegant solution:

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x==y || elem x ys
```

As in imperative languages, the `(||)` operator will only evaluate its second argument if the first one is `False`. So you don't need to worry about inefficiency in this solution, because the recursive call is only made if needed.

If you are particularly observant, you might now notice that the definition of `elem` looks a lot like something that could be replaced by `foldr` with appropriate arguments and indeed, you can write something like the following:

```
elem :: Eq a => a -> [a] -> Bool
elem x = foldr (\y r -> x==y || r) False
```

8.2 Higher-order functions

Ex45 For each of the following statements, discuss with someone (friend, tutor, rubber duck, etc.) whether it is true or false:

1. A function of type `a -> b -> c` returns a function.
   Solution: This is true. Function types associate to the right, so that this type really means `a -> (b -> c)`: a function from some value of type `a` to a function of type `b -> c`. Note that there are no functions whose type is exactly `a -> b -> c` however.

2. A function of type `(a -> b) -> Int` returns a function.
   Solution: This is false. The type shows that the function always returns a value of type `Int`.

3. A function of type `(Int, Bool) -> Char` is higher-order.
   Solution: This is false. A function is higher-order if it takes a function as argument or if it returns a function. Functions of the type shown above do neither.

4. A function of type `a -> a` can be a higher-order function.
   Solution: This is true. The type variable `a` can be instantiated with any other type, including function types. The identity function `id` is the only function of this type:

```
id :: a -> a
id x = x
```

Applying `id` to another function just returns that function. For example, `id map` evaluates to `map`.

8.3 Equational reasoning

Induction on natural numbers

Ex90 For this task, we need to prove the following property of addition which states that adding the successor of zero to some number n is the same as the successor of n:

$$\forall n :: Nat.S\ n = add\ (S\ Z)\ n$$

The proof can simply be done by rewriting one side of the equation to the other:

$$\begin{aligned} & add\ (S\ Z)\ n \\ = & \quad \{ \quad \text{applying } add \quad \} \\ & S\ (add\ Z\ n) \\ = & \quad \{ \quad \text{applying } add \quad \} \\ & S\ n \end{aligned}$$

Ex91 For this task, we need to prove another property of addition which states that the adding the successor of some number n to some other number m is the same as adding n to the successor of m:

$$\forall n\ m :: Nat.add\ (S\ n)\ m = add\ n\ (S\ m)$$

This proof is by induction on n. First we prove the base case for Z:

$$\begin{aligned} & add\ (S\ Z)\ m \\ = & \quad \{ \quad \text{property proved in } \textbf{Ex90} \text{ that } \forall n.add\ (S\ Z)\ n = S\ n \quad \} \\ & S\ m \\ = & \quad \{ \quad \text{unapplying } add \quad \} \\ & add\ Z\ (S\ m) \end{aligned}$$

Then we prove the inductive case for $S\ n$. Our induction hypothesis is that:

$$\forall m :: Nat.add\ (S\ n)\ m = add\ n\ (S\ m)$$

As usual, we pick one side of the equation and rewrite it to the other side:

$$
\begin{aligned}
& \mathit{add}\ (S\ (S\ n))\ m \\
= & \quad \{ \quad \text{applying } \mathit{add} \quad \} \\
& S\ (\mathit{add}\ (S\ n)\ m) \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
& S\ (\mathit{add}\ n\ (S\ m)) \\
= & \quad \{ \quad \text{unapplying } \mathit{add} \quad \} \\
& \mathit{add}\ (S\ n)\ (S\ m)
\end{aligned}
$$

Ex92 For this task, we need to prove that addition is commutative:

$$\forall n\ m :: \mathit{Nat}.\mathit{add}\ n\ m = \mathit{add}\ m\ n$$

As there is no obvious way for us to start rewriting either side of the equation, we perform induction on n. First we prove the base case for Z:

$$
\begin{aligned}
& \mathit{add}\ Z\ m \\
= & \quad \{ \quad \text{applying } \mathit{add} \quad \} \\
& m \\
= & \quad \{ \quad \text{right identity of } \mathit{add}\text{, proved in Section 3.1} \quad \} \\
& \mathit{add}\ m\ Z
\end{aligned}
$$

Now we can move on to the inductive step for $S\ n$, for which our induction hypothesis is as follows:

$$\forall m :: \mathit{Nat}.\mathit{add}\ n\ m = \mathit{add}\ m\ n$$

Using this, we can then conclude the proof:

$$
\begin{aligned}
& \mathit{add}\ (S\ n)\ m \\
= & \quad \{ \quad \text{applying } \mathit{add} \quad \} \\
& S\ (\mathit{add}\ n\ m) \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
& S\ (\mathit{add}\ m\ n) \\
= & \quad \{ \quad \text{unapplying } \mathit{add} \quad \} \\
& \mathit{add}\ (S\ m)\ n \\
= & \quad \{ \quad \text{property proved for } \textbf{Ex91} \quad \} \\
& \mathit{add}\ m\ (S\ n)
\end{aligned}
$$

Induction on lists

Ex93 For this task, we need to prove the following property about $+\!\!+$ which says that its left identity is $[\,]$:

$$\forall xs :: [a]\,.\quad [\,] +\!\!+ xs = xs$$

We can prove this property easily just by rewriting one side of the equation into the other:

$$\begin{aligned} & [\,] +\!\!+ xs \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & xs \end{aligned}$$

Ex94 For this task, we need to prove the following property about $+\!\!+$ which says that its right identity is $[\,]$:

$$\forall xs :: [a]\,.\quad xs +\!\!+ [\,] = xs$$

This proof is by induction on *xs*. The base case is for the empty list:

$$\begin{aligned} & [\,] +\!\!+ [\,] \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & [\,] \end{aligned}$$

For the inductive case where we prove the property for $x : xs$, our induction hypothesis is:

$$xs +\!\!+ [\,] = xs$$

The proof is as follows:

$$\begin{aligned} & (x : xs) +\!\!+ [\,] \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & x : (xs +\!\!+ [\,]) \\ = & \quad \{ \quad \text{induction hypothesis} \quad \} \\ & x : xs \end{aligned}$$

Ex95 For this task, we need to prove that $+\!\!+$ is associative:

$$\forall xs\ ys\ zs :: [a]\,.\quad xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

This proof is by induction on *xs* and the base case is as usual for the empty list:

$$\begin{aligned} & [\,] +\!\!+ (ys +\!\!+ zs) \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & ys +\!\!+ zs \\ = & \quad \{ \quad \text{unapplying } +\!\!+ \quad \} \\ & ([\,] +\!\!+ ys) +\!\!+ zs \end{aligned}$$

We can now move on to the inductive step. Our induction hypothesis is:

$$\forall ys\ zs :: [a]\ .\quad xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

The inductive step is for $x : xs$:

$$\begin{aligned} & (x : xs) +\!\!+ (ys +\!\!+ zs) \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & x : (xs +\!\!+ (ys +\!\!+ zs)) \\ = & \quad \{ \quad \text{induction hypothesis} \quad \} \\ & x : ((xs +\!\!+ ys) +\!\!+ zs) \\ = & \quad \{ \quad \text{unapplying } +\!\!+ \quad \} \\ & (x : (xs +\!\!+ ys)) +\!\!+ zs \\ = & \quad \{ \quad \text{unapplying } +\!\!+ \quad \} \\ & ((x : xs) +\!\!+ ys) +\!\!+ zs \end{aligned}$$

Ex96 For this task, we need to prove that *reverse*, given a singleton list, evaluates to the same singleton list:

$$\forall x :: a\ .\quad reverse\ [x] = [x]$$

We can prove this simply by rewriting one side of the equation:

$$\begin{aligned} & reverse\ [x] \\ = & \quad \{ \quad \text{applying } reverse \quad \} \\ & reverse\ [\,] +\!\!+ [x] \\ = & \quad \{ \quad \text{applying } reverse \quad \} \\ & [\,] +\!\!+ [x] \\ = & \quad \{ \quad \text{applying } +\!\!+ \quad \} \\ & [x] \end{aligned}$$

Ex97 For this task, we need to prove that *reverse* distributes over $+\!\!+$:

$$\forall xs\ ys :: [a]\ .\quad reverse\ (xs +\!\!+ ys) = reverse\ ys +\!\!+ reverse\ xs$$

This proof is by induction on *xs* and the base case is, as usual, for the empty list:

$$
\begin{aligned}
& \mathit{reverse}\ ([\,] \mathbin{+\!\!\!+} \mathit{ys}) \\
= \quad & \{ \quad \text{applying } \mathbin{+\!\!\!+} \quad \} \\
& \mathit{reverse}\ \mathit{ys} \\
= \quad & \{ \quad \text{property proved in } \textbf{Ex94} \quad \} \\
& \mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} [\,] \\
= \quad & \{ \quad \text{unapplying } \mathit{reverse} \quad \} \\
& \mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} \mathit{reverse}\ [\,]
\end{aligned}
$$

Our induction hypothesis is:

$$\forall \mathit{ys} :: [a]\ . \quad \mathit{reverse}\ (\mathit{xs} \mathbin{+\!\!\!+} \mathit{ys}) = \mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} \mathit{reverse}\ \mathit{xs}$$

The inductive step for $x : xs$ is:

$$
\begin{aligned}
& \mathit{reverse}\ ((x : \mathit{xs}) \mathbin{+\!\!\!+} \mathit{ys}) \\
= \quad & \{ \quad \text{applying } \mathbin{+\!\!\!+} \quad \} \\
& \mathit{reverse}\ (x : (\mathit{xs} \mathbin{+\!\!\!+} \mathit{ys})) \\
= \quad & \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& \mathit{reverse}\ (\mathit{xs} \mathbin{+\!\!\!+} \mathit{ys}) \mathbin{+\!\!\!+} [x] \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& (\mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} \mathit{reverse}\ \mathit{xs}) \mathbin{+\!\!\!+} [x] \\
= \quad & \{ \quad \text{associativity of } \mathbin{+\!\!\!+} \text{ proved for } \textbf{Ex95} \quad \} \\
& \mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} (\mathit{reverse}\ \mathit{xs} \mathbin{+\!\!\!+} [x]) \\
= \quad & \{ \quad \text{unapplying } \mathit{reverse} \quad \} \\
& \mathit{reverse}\ \mathit{ys} \mathbin{+\!\!\!+} \mathit{reverse}\ (x : \mathit{xs})
\end{aligned}
$$

Ex98 For this task, we need to prove that the reverse of the reverse of a list is just the list we start with:

$$\forall \mathit{xs} :: [a]\ . \quad \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs}$$

The proof is by induction on *xs*. As usual, the base case is for the empty list:

$$
\begin{aligned}
& \mathit{reverse}\ (\mathit{reverse}\ [\,]) \\
= \quad & \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& \mathit{reverse}\ [\,] \\
= \quad & \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& [\,]
\end{aligned}
$$

This concludes the proof for the base case. Now we need to move on with the inductive case for $x : xs$. Our induction hypothesis is:

$$\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs}$$

The proof for the inductive case is then:

$$\begin{aligned}
& \mathit{reverse}\ (\mathit{reverse}\ (x : \mathit{xs})) \\
= & \quad \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} [x]) \\
= & \quad \{ \quad \mathit{reverse} \text{ distributes over } \mathbin{+\!\!+}, \text{ proved for } \textbf{Ex97} \quad \} \\
& \mathit{reverse}\ [x] \mathbin{+\!\!+} \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \\
= & \quad \{ \quad \mathit{reverse} \text{ of a singleton list, proved for } \textbf{Ex96} \quad \} \\
& [x] \mathbin{+\!\!+} \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
& [x] \mathbin{+\!\!+} \mathit{xs} \\
= & \quad \{ \quad \text{applying } \mathbin{+\!\!+} \quad \} \\
& x : ([\,] \mathbin{+\!\!+} \mathit{xs}) \\
= & \quad \{ \quad \text{applying } \mathbin{+\!\!+} \quad \} \\
& x : \mathit{xs}
\end{aligned}$$

This concludes the proof.

Constructive induction

Ex99 For this task, we simply need to simplify the following expression until we cannot simplify it any further. The resulting expression is then used as the RHS of the first equation for the *rev* function we are trying to define:

$$\begin{aligned}
& \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \\
= & \quad \{ \quad \text{applying } \mathit{reverse} \quad \} \\
& [\,] \mathbin{+\!\!+} \mathit{ys} \\
= & \quad \{ \quad \text{applying } \mathbin{+\!\!+} \quad \} \\
& \mathit{ys}
\end{aligned}$$

This expression cannot be simplified any further and we therefore use it as the RHS of our definition for the new *rev* function:

```
rev :: [a] -> [a] -> [a]
rev []     ys = ys
rev (x:xs) ys = ???
```

Ex100 For this task, we simply need to simplify the following expression until we cannot simplify it any further. The resulting expression is then used as the RHS of the first equation for the *rev* function we are trying to define. We also have access to the following induction hypothesis:

$$\forall ys :: [a] \,.\quad rev\ xs\ ys = reverse\ xs +\!\!+\, ys$$

Let us begin to simplify the expression:

$$\begin{aligned}
& reverse\ (x : xs) +\!\!+\, ys \\
= \quad & \{ \quad \text{applying } reverse \quad \} \\
& (reverse\ xs +\!\!+\, [x]) +\!\!+\, ys \\
= \quad & \{ \quad \text{associativity of } +\!\!+\text{, proved for } \textbf{Ex95} \quad \} \\
& reverse\ xs +\!\!+\, ([x] +\!\!+\, ys) \\
= \quad & \{ \quad \text{applying } +\!\!+ \quad \} \\
& reverse\ xs +\!\!+\, (x : ([\,] +\!\!+\, ys)) \\
= \quad & \{ \quad \text{applying } +\!\!+ \quad \} \\
& reverse\ xs +\!\!+\, (x : ys) \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& rev\ xs\ (x : ys)
\end{aligned}$$

This expression cannot be simplified any further and does no longer contain references to *reverse* or $+\!\!+$, so we use it as the RHS of the second equation of our definition for *rev*:

```
rev :: [a] -> [a] -> [a]
rev []     ys = ys
rev (x:xs) ys = rev xs (x:ys)
```

This function is much more efficient than our old definition of *reverse* and more general. We can restore the behaviour of *reverse* by defining it in terms of *rev* as follows:

```
reverse :: [a] -> [a]
reverse xs = rev xs []
```

8.4 Functors

Ex109 Let us prove the two functor laws for the `Identity` type. First up:

$$\forall x :: a \,.\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Identity}\ x) = \mathit{id}\ (\mathit{Identity}\ x)$$

We can prove this simply by rewriting the equations:

$$\begin{array}{ll}
& \mathit{fmap}\ \mathit{id}\ (\mathit{Identity}\ x) \\
= & \{\ \text{applying } \mathit{fmap}\ \} \\
& \mathit{Identity}\ (\mathit{id}\ x) \\
= & \{\ \text{applying } \mathit{id}\ \} \\
& \mathit{Identity}\ x \\
= & \{\ \text{unapplying } \mathit{id}\ \} \\
& \mathit{id}\ (\mathit{Identity}\ x)
\end{array}$$

Next up is the fusion proof:

$$\forall x :: a, f :: b \to c, g :: a \to b \,.\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Identity}\ x) = (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Identity}\ x)$$

The proof for this is again accomplished by just rewriting one side of the equation:

$$\begin{array}{ll}
& \mathit{fmap}\ (f \circ g)\ (\mathit{Identity}\ x) \\
= & \{\ \text{applying } \mathit{fmap}\ \} \\
& \mathit{Identity}\ ((f \circ g)\ x) \\
= & \{\ \text{applying } \circ\ \} \\
& \mathit{Identity}\ (f\ (g\ x)) \\
= & \{\ \text{unapplying } \mathit{fmap}\ \} \\
& \mathit{fmap}\ f\ (\mathit{Identity}\ (g\ x)) \\
= & \{\ \text{unapplying } \mathit{fmap}\ \} \\
& \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Identity}\ x)) \\
= & \{\ \text{unapplying } \circ\ \} \\
& (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Identity}\ x)
\end{array}$$

Ex111 Let us prove the two functor laws for the `Const` type. First up:

$$\forall x :: v \,.\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Const}\ x) = \mathit{id}\ (\mathit{Const}\ x)$$

We can prove this simply by rewriting the equations:

$$\mathit{fmap}\ \mathit{id}\ (\mathit{Const}\ x)$$

$$\begin{aligned}
&= \quad \{ \text{ applying } \mathit{fmap} \ \} \\
&\quad \mathit{Const}\ x \\
&= \quad \{ \text{ unapplying } \mathit{id} \ \} \\
&\quad \mathit{id}\ (\mathit{Const}\ x)
\end{aligned}$$

Next up is the fusion proof:

$$\forall x :: v, f :: b \to c, g :: a \to b\ .\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Const}\ x) = (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Const}\ x)$$

The proof for this is again accomplished by just rewriting one side of the equation:

$$\begin{aligned}
&\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Const}\ x) \\
&= \quad \{ \text{ applying } \mathit{fmap} \ \} \\
&\quad \mathit{Const}\ x \\
&= \quad \{ \text{ unapplying } \mathit{fmap} \ \} \\
&\quad \mathit{fmap}\ f\ (\mathit{Const}\ x) \\
&= \quad \{ \text{ unapplying } \mathit{fmap} \ \} \\
&\quad \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Const}\ x)) \\
&= \quad \{ \text{ unapplying } \circ \ \} \\
&\quad (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Const}\ x)
\end{aligned}$$

Ex113 Let us prove the two functor laws for the `Point` type. First up:

$$\forall x\ y :: a\ .\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Point}\ x\ y) = \mathit{id}\ (\mathit{Point}\ x\ y)$$

We can prove this simply by rewriting the equations:

$$\begin{aligned}
&\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Point}\ x\ y) \\
&= \quad \{ \text{ applying } \mathit{fmap} \ \} \\
&\quad \mathit{Point}\ (\mathit{id}\ x)\ (\mathit{id}\ y) \\
&= \quad \{ \text{ applying } \mathit{id} \text{ twice } \} \\
&\quad \mathit{Point}\ x\ y \\
&= \quad \{ \text{ unapplying } \mathit{id} \ \} \\
&\quad \mathit{id}\ (\mathit{Point}\ x\ y)
\end{aligned}$$

Next up is the fusion proof:

$$\forall x\ y :: a, f :: b \to c, g :: a \to b\ .\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Point}\ x\ y) = (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Point}\ x\ y)$$

The proof for this is again accomplished by just rewriting one side of the equation:

$$\mathit{fmap}\ (f \circ g)\ (\mathit{Point}\ x\ y)$$

$$
\begin{aligned}
&= \quad \{ \quad \text{applying } \mathit{fmap} \quad \} \\
&\quad \mathit{Point}\ ((f \circ g)\ x)\ ((f \circ g)\ y) \\
&= \quad \{ \quad \text{applying } \circ \text{ twice} \quad \} \\
&\quad \mathit{Point}\ (f\ (g\ x))\ (f\ (g\ y)) \\
&= \quad \{ \quad \text{unapplying } \mathit{fmap} \quad \} \\
&\quad \mathit{fmap}\ f\ (\mathit{Point}\ (g\ x)\ (g\ y)) \\
&= \quad \{ \quad \text{unapplying } \mathit{fmap} \quad \} \\
&\quad \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Point}\ x\ y)) \\
&= \quad \{ \quad \text{unapplying } \circ \quad \} \\
&\quad (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Point}\ x\ y)
\end{aligned}
$$

Ex117 Let us prove the two functor laws for the `Compose` type. First up:

$$\forall x :: f\ (g\ a).\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Compose}\ x) = \mathit{id}\ (\mathit{Compose}\ x)$$

We can prove this again simply by rewriting the equations:

$$
\begin{aligned}
&\quad \mathit{fmap}\ \mathit{id}\ (\mathit{Compose}\ x) \\
&= \quad \{ \quad \text{applying } \mathit{fmap} \quad \} \\
&\quad \mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{id})\ x) \\
&= \quad \{ \quad \text{the type } g \text{ is a functor, therefore the identity law holds} \quad \} \\
&\quad \mathit{Compose}\ (\mathit{fmap}\ \mathit{id}\ x) \\
&= \quad \{ \quad \text{the type } f \text{ is a functor, therefore the identity law holds} \quad \} \\
&\quad \mathit{Compose}\ (\mathit{id}\ x) \\
&= \quad \{ \quad \text{applying } \mathit{id} \quad \} \\
&\quad \mathit{Compose}\ x \\
&= \quad \{ \quad \text{unapplying } \mathit{id} \quad \} \\
&\quad \mathit{id}\ (\mathit{Compose}\ x)
\end{aligned}
$$

Next up is the fusion proof:

$$
\begin{aligned}
&\forall x :: f\ (g\ a), f :: b \to c, g :: a \to b\ . \\
&\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Compose}\ x) = (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Compose}\ x)
\end{aligned}
$$

The proof for this is again accomplished by just rewriting one side of the equation:

$$
\begin{aligned}
&\quad \mathit{fmap}\ (f \circ g)\ (\mathit{Compose}\ x) \\
&= \quad \{ \quad \text{applying } \mathit{fmap} \quad \} \\
&\quad \mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ (f \circ g))\ x) \\
&= \quad \{ \quad \text{the type } g \text{ is a functor, therefore the fusion law holds} \quad \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ x) \\
= & \quad \{ \quad \text{the type } f \text{ is a functor, therefore the fusion law holds} \quad \} \\
& \mathit{Compose}\ ((\mathit{fmap}\ (\mathit{fmap}\ f) \circ \mathit{fmap}\ (\mathit{fmap}\ g))\ x) \\
= & \quad \{ \quad \text{applying } \circ \quad \} \\
& \mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ (\mathit{fmap}\ (\mathit{fmap}\ g)\ x)) \\
= & \quad \{ \quad \text{unapplying } \mathit{fmap} \quad \} \\
& \mathit{fmap}\ f\ (\mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ g)\ x)) \\
= & \quad \{ \quad \text{unapplying } \mathit{fmap} \quad \} \\
& \mathit{fmap}\ f\ (\mathit{fmap}\ g\ (\mathit{Compose}\ x)) \\
= & \quad \{ \quad \text{unapplying } \circ \quad \} \\
& (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ (\mathit{Compose}\ x)
\end{aligned}
$$

λ.9

Haskell Prelude

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max             :: a -> a -> Bool

  min x y | x <= y     = x
          | otherwise = y

  max x y | x <= y     = y
          | otherwise = x
```

```
class Enum a where
  succ           :: a -> a
  pred           :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
```

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs           :: a -> a
  signum        :: a -> a
  fromInteger   :: Integer -> a
```

```
class Enum a => Integral a where
  quot      :: a -> a -> a
  rem       :: a -> a -> a
  div       :: a -> a -> a
  mod       :: a -> a -> a
  quotRem   :: a -> a -> (a, a)
  divMod    :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

```
class Num a => Fractional a where
  (/)          :: a -> a -> a
  recip        :: a -> a
  fromRational :: Rational -> a
```

```
data Int = ...
  deriving ( Eq, Ord, Show, Read
           , Num, Integral )
```

```
data Integer = ...
  deriving ( Eq, Ord, Show, Read
           , Num, Integral )
```

```
data Float = ...
  deriving ( Eq, Ord, Show, Read
           , Num, Fractional )
```

```
data Double = ...
  deriving ( Eq, Ord, Show, Read
           , Num, Fractional )
```

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

```
odd :: Integral a => a -> Bool
odd = not . even
```

```
class Show a where
  show :: a -> String
```

```
class Read a where
  read :: String -> a
```

```
class Foldable t where
  foldr   ::
    (a -> b -> b) -> b -> t a -> b
  foldl   ::
    (b -> a -> b) -> b -> t a -> b
  foldr1  :: (a -> a -> a) -> t a -> a
  foldl1  :: (a -> a -> a) -> t a -> a

  null :: t a -> Bool
  null = foldr (\_ _ -> False) True

  length :: t a -> Int
  length = foldr (\x r -> 1 + r) 0

  elem :: Eq a -> a -> t a -> Bool
  elem x =
    foldr (\y r -> x==y || r) False

  maximum :: Ord a => t a -> a
  maximum = foldl1 max

  minimum :: Ord a => t a -> a
  minimum = foldl1 min

  sum :: Num a => t a -> a
  sum = foldl (+) 0

  product :: Num a => t a -> a
  product = foldl (*) 1
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

```
id :: a -> a
id x = x
```

```
const :: a -> b -> a
const x _ = x
```

```
($!) :: (a -> b) -> a -> b
f $! x = ...
```

Semigroups and Monoids

```
class Semigroup a where
  (<>) :: a -> a -> a
```

```
class Semigroup a => Monoid a where
  mempty :: a
```

Left identity	$\mathit{mempty} <> x$	$=$	x
Right identity	$x <> \mathit{mempty}$	$=$	x
Associativity	$(x <> y) <> z$	$=$	$x <> (y <> z)$

Functors

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Identity	$\mathit{fmap}\ \mathit{id}$	$=$	id
Fusion	$\mathit{fmap}\ (f \circ g)$	$=$	$\mathit{fmap}\ f \circ \mathit{fmap}\ g$

Applicatives

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Identity	$\mathit{pure}\ \mathit{id} <*> v = v$
Homomorphism	$\mathit{pure}\ f <*> \mathit{pure}\ x = \mathit{pure}\ (f\ x)$
Interchange	$u <*> \mathit{pure}\ y = \mathit{pure}\ (\$\ y) <*> u$
Composition	$\mathit{pure}\ (\circ) <*> u <*> v <*> w = u <*> (v <*> w)$

Monads

```
class Applicative m => Monad m where
  return :: a -> m a
  return = pure

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

Left identity	$\mathit{return}\ a \gg= f$	$=$	$f\ a$
Right identity	$m \gg= \mathit{return}$	$=$	m
Associativity	$(m \gg= f) \gg= g$	$=$	$m \gg= (\backslash x \to f\ x \gg= g)$

Booleans

```
data Bool = True | False
  deriving ( Bounded, Enum, Eq, Ord
           , Read, Show )
```

```
not :: Bool -> Bool
not True  = False
not False = True
```

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
(||) :: Bool -> Bool -> Bool
False || False = False
_     || _     = True
```

```
and :: Foldable t => t Bool -> Bool
and = foldr (&&) True
```

```
or :: Foldable t => t Bool -> Bool
or = foldr (||) False
```

```
all :: Foldable t =>
    (a -> Bool) -> t a -> Bool
all p = and . foldr (\x xs -> p x : xs) []
```

```
any :: Foldable t =>
    (a -> Bool) -> t a -> Bool
any p = or . foldr (\x xs -> p x : xs) []
```

```
otherwise :: Bool
otherwise = True
```

Characters

```
data Char = ...
```

```
type String = [Char]
```

```
isLower :: Char -> Bool
isLower c = c >= 'a' && c <= 'z'
```

```
isUpper :: Char -> Bool
isUpper c = c >= 'A' && c <= 'Z'
```

```
isAlpha :: Char -> Bool
isAlpha c = isLower c || isUpper c
```

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

```
isAlphaNum :: Char -> Bool
isAlphaNum c = isAlpha c || isDigit c
```

```
isSpace :: Char -> Bool
isSpace c = c `elem` " \t\n"
```

```
ord :: Char -> Int
ord c = ...
```

```
chr :: Int -> Char
chr n = ...
```

```
digitToInt :: Char -> Int
digitToInt c | isDigit c = ord c - ord '0'
```

```
intToDigit :: Int -> Char
intToDigit n
  | n >= 0 && n <= 9 = chr (ord '0' + n)
```

```
toLower :: Char -> Char
toLower c
  | isUpper c =
      chr (ord c - ord 'A' + ord 'a')
  | otherwise = c
```

```
toUpper :: Char -> Char
toUpper c
  | isLower c =
      chr (ord c - ord 'a' + ord 'A')
  | otherwise = c
```

Lists

```
data [a] = [] | (:) a [a]
  deriving (Eq, Ord, Show, Read)
```

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

```
instance Applicative [] where
  pure x = [x]

  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
instance Monad [] where
  xs >>= f = [y | x <- xs, y <- f x]
```

```
instance Foldable [] where
  foldr _ v []     = v
  foldr f v (x:xs) = f x (foldr f v xs)

  foldr1 _ [x]    = x
  foldr1 f (x:xs) = f x (foldr1 f xs)

  foldl _ v []     = v
  foldl f v (x:xs) = foldl f (f v x) xs

  foldl1 f (x:xs) = foldl f x xs
```

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

```
last :: [a] -> a
last [x]    = x
last (x:xs) = last xs
```

```
init :: [a] -> [a]
init [_]    = []
init (x:xs) = x : init xs
```

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

```
lookup :: Eq k => k -> [(k,v)] -> Maybe v
lookup x [] = Nothing
lookup x ((y,v):ys)
  | x == y    = Just v
  | otherwise = lookup x ys
```

```
(!!) :: [a] -> Int -> a
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n-1)
```

```
take :: Int -> [a] -> [a]
take 0 _      = []
take n []     = []
take n (x:xs) = x : take (n-1) xs
```

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
  | p x       = x : takeWhile p xs
  | otherwise = []
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs)
  | p x       = dropWhile p xs
  | otherwise = x : xs
```

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span p xs =
  (takeWhile p xs, dropWhile p xs)
```

```
repeat :: a -> [a]
repeat x = xs where xs = x : xs
```

```
replicate :: Int -> a -> [a]
replicate n = take n . repeat
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

```
reverse :: [a] -> [a]
reverse = foldl (\xs x -> x : xs) []
```

```
subsequences :: [a] -> [[a]]
subsequences []     = [[]]
subsequences (x:xs) = ys ++ map (x:) ys
  where ys = subsequences xs
```

```
nub :: Eq a => [a] -> [a]
nub []     = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

```
delete :: Eq a => a -> [a] -> [a]
delete _ []     = []
delete x (y:ys)
  | x == y    = ys
  | otherwise = y : delete x ys
```

Maybe

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)
```

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

```
instance Applicative Maybe where
  pure x = Just x

  Nothing  <*> _ = Nothing
  (Just f) <*> y = fmap f y
```

```
instance Monad Maybe where
  Nothing  >>= f = Nothing
  (Just x) >>= f = f x
```

Either

```
data Either a b = Left a | Right b
```

```
instance Functor (Either e) where
  fmap f (Left x)  = Left x
  fmap f (Right y) = Right (f y)
```

```
instance Applicative (Either e) where
  pure = Right

  Left e  <*> _ = Left e
  Right f <*> x = fmap f x
```

```
instance Monad (Either e) where
  Left e >>= _  = Left e
  Right x >>= f = f x
```

Tuples

All types of tuples are instances of *Eq*, *Ord*, *Show*, *Read* provided that their components are also instances of those type classes.

```
fst :: (a, b) -> a
fst (x,y) = x
```

```
snd :: (a, b) -> b
snd (x,y) = y
```

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

IO

```
data IO a = ...
```

```
instance Functor IO where ...
instance Applicative IO where ...
instance Monad IO where ...
```

```haskell
getChar :: IO Char
getChar = ...
```

```haskell
getLine :: IO String
getLine = ...
```

```haskell
putChar :: Char -> IO ()
putChar c = ...
```

```haskell
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = putChar x >> putStr xs
```

```haskell
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

```haskell
print :: Show a => a -> IO ()
print = putStrLn . show
```

Type-level programming

The kind of types is denoted as `*` or `Type`.

```haskell
data Nat = Zero | Succ Nat
```

Bibliography

Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2016.

Kiselyov, Oleg & Jones, Simon Peyton & Shan, Chung-chieh. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

Lipovača, Miran. *Learn You a Haskell for Great Good!: A Beginner's Guide*. no starch press, 2011.

Okasaki, Chris. *Purely functional data structures*. Cambridge University Press, 1999.

Peyton Jones, Simon. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

Peyton Jones, Simon. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Hoare, CAR & Broy, M & Steinbrueggen, R, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.

Wadler, Philip. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

Yorgey, Brent A & Weirich, Stephanie & Cretin, Julien & Peyton Jones, Simon & Vytiniotis, Dimitrios & Magalhães, José Pedro. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.