IT5100A

Industry Readiness: Typed Functional Programming

*by **Foo** Yong Qi*

GitHub · Report Bug · Request Feature

This is a set of lecture notes for students enrolled in IT5100A—Industry Readiness: Typed Functional Programming in NUS SoC.

About IT5100A

Typed functional programming are becoming more widely adopted in industry, as can be seen in the success of a number of advanced programming languages, such as OCaml, Haskell and Scala 3. These advanced languages offer a range of expressive features to allow robust, reusable and high-performing software codes to be safely and rapidly developed. This course will cover key programming techniques of typed functional programming that are becoming widely adopted, such as strong typing, code composition and abstraction, effect handlers, and safe techniques for asynchronous and concurrent programming.

About These Notes

I hope that these notes can be used as good supplementary material for those looking to learn the concepts of Typed Functional Programming in more detail. Each of these chapters comes with exercises in Python and Haskell so that you're able to replicate some of the ideas from purely-functional languages in general-purpose multi-paradigm languages.

Therefore, to avoid confusion, code blocks are annotated with the logo of the target programming language on the left. Examples below. (Readers on mobile might have to rotate their phones to landscape to view the logos.)

Python

```
this = 'is some Python code'
```

Haskell

```haskell
this :: String
this = "is some Haskell code"
```

Java

```java
class This {
    public static void main(String[] args) {
        System.out.println("is some Java code");
    }
}
```

Lean 4

```
def this: String := "is some Lean 4 code"
```

Updates

This work is incomplete, and therefore will be regularly updated. As such, please run a hard refresh (`Ctrl + F5`) every time you visit the page.

A badge is shown at the beginning of every page describing when it was last updated. The badge looks like this:

Ensure that the badge displays the expected date of the last update.

Contributing

This project is a single-author text, and is incomplete. Thus, this project is not open to pull requests without prior agreement. However, please feel free to improve the quality of this content by submitting bug reports and feature requests. All your contributions other than by the author will be considered a donation of your work to this project, and you are not considered an author or owner of the content once they have been incorporated.

Please submit all requests for content and bugs either as a GitHub issue or contact the author directly.

Contributors

License

Contact

Author: Foo Yong Qi - yongqi@nus.edu.sg

Release History

2024

Date	Description
26 Oct	Writeup on Concurrent and Parallel Programming, excluding exercises
13 Oct	Writeup on Monads, excluding exercises
10 Oct	Writeup on the existential typeclass pattern
28 Sep	• Additional writeups and bug fixes in the existing chapters • Bug fixes in operator highlighting in code blocks in light modes. • Recap on first-class functions and lambda calculus.

Date	Description
	• Solutions to exercises for the first four chapters.
26 Sep	The first draft of these notes have been released with the first four chapters completed.

LAST UPDATED **26 OCT 2024**

In this chapter, we go through some of the usual administrivia of this course, and proceed to discuss some core ideas of Functional Programming (FP) in different settings, some which should be unfamiliar to you.

Readers who find some of the concepts in Chapter 1.2 (Functional Programming) challenging or unfamiliar can revisit these ideas in Chapter 8 (Recap of Concepts) before proceeding.

LAST UPDATED 26 OCT 2024

Course Administration

Course Coordinator

Foo Yong Qi

Instructor & Ph.D. Student

Email: yongqi@nus.edu.sg

Course Outline

- Course Introduction
  - Course Administration
  - Functional Programming
  - Introduction to Haskell
- Types
  - Types and Type Systems
  - Polymorphism
  - Algebraic Data Types
  - Pattern Matching
- Typeclasses
  - What Are Typeclasses?
  - Important Typeclasses
  - Typeclasses and Typeclass Instances
- Railway Pattern
  - Functors
  - Applicative Functors
  - Validation
  - Monads
- Monads
  - Commonly-Used Monads
  - Monad Transformers
- Concurrent Programming
  - Concurrent Programming with Threads
  - Parallel Programming
  - Software Transactional Memory
- Course Conclusion

Graded Items

Item	Weightage
Assignment 1	20%
Assignment 2	20%
Assignment 3	20%
Practical Exam	40%

The Practical Exam is planned to be during the last lecture.

Plagiarism Notice

Assignments are on programming... standard plagiarism rules apply.

No code sharing!

- ChatGPT (and similar tools) is allowed for **learning only**
- Using LLMs to generate code is **not allowed**
- NUS takes a strict view of plagiarism and cheating
- Disciplinary action will be taken against students who violate NUS Student Code of Conduct
- No part of your assignment can come from any other source
- No discussion and sharing of solutions during exams

LAST UPDATED 26 OCT 2024

Functional Programming

Functional Programming (FP) is a *declarative programming paradigm* where *functions* take centre stage. As a recap from IT5001, you might have learnt that programming paradigms are schools of thought for writing programs. IT5001 has very likely exposed you to *imperative* paradigms like *procedural* and *Object-Oriented Programming*. The following table shows other popular programming paradigms:

Imperative	Declarative
Procedural	Logic
Object-Oriented	Functional

Object-Oriented Programming (OOP) has four principles as you might recall: *Abstraction*, *Inheritance*, *Encapsulation* and *Polymorphism*.[1] Functional Programming, on the other hand, is centered around the following principles, which really are just principles of mathematical functions and the λ *calculus*:[2]

- Immutability
- Pure Functions
- Recursion
- Types
- First-Class Functions

Let's briefly describe what these principles entail.

Immutability

The idea of *immutability* is simple—only use **immutable** data. For example, the following program fragment does not perform any mutation, not even on the variables:

```
def add_one(fraction):
    """fraction is a tuple of (numerator, denominator)"""
    old_num, den = fraction
    num = old_num + den
    return (num, den)

my_fraction = (3, 2)
new_fraction = add_one(my_fraction)

print(new_fraction) # (5, 2)
print(my_fraction) # (3, 2)
```

The fact that the program does not perform any mutation makes this very similar to mathematical functions where mathematical objects are seen as values instead of references to cells that can be changed. This makes reasoning about any of the variables, objects and functions incredibly simple.

Overall, immutability forces us to be disciplined with **state**. Contrast this with using **mutable** data structures and variables, such as in the following program fragment:

```
def f(ls):
  ls[0] = 4
  return ls

my_ls = [1, 2, 3]
print(f(my_ls)) # [4, 2, 3]
print(my_ls) # [4, 2, 3]
```

This is one of the classic examples of the problems with mutability—it is not at all clear whether passing a list into a function will preserve the state of the list. Because lists are mutable, we have no guarantee that functions or any operation will not cause the *side-effect* of mutation (accidental or intentional).

Pure Functions

Just like mathematical functions, functions (in programming) should be *pure*. Pure functions really look like mathematical functions, for example, f below:

$$\mathrm{f} : \mathbb{N} \to \mathbb{N}$$

$$\mathrm{f}(\mathrm{x}) = \mathrm{x}^2 + 2\mathrm{x} + 3$$

An equivalent implementation in Python would look like:

```
def f(x):
  return x ** 2 + 2 * x + 3
```

Pure functions **only receive input and return output**. They do not produce side effects, and do not depend on external state. An example of this is as follows:

```python
# Python
def double(ls):
  return [i * 2 for i in ls]

x = [1, 2, 3]

print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
print(double(x)) # ...
# ...
```

Notice that the `double` function is pure! In this example, `double(x)` evaluates to `[2, 4, 6]`; thus, `double(x)` and `[2, 4, 6]` are the **same**! This property of pure functions is known as *referential transparency*, and makes reasoning about and optimizing programs much more straightforward.

Contrast the behaviour of pure functions with that of impure functions:

```python
def f():
  global ls
  x = ls # use of global variable
  addend = x[-1] + 1
  x.append(addend) # is there a side-effect?
  ls = x + [addend + 1] # mutate global variable
  return ls

ls = [1, 2, 3]
x = ls

print(f()) # [1, 2, 3, 4, 5]
print(ls) # [1, 2, 3, 4, 5]
print(x) # [1, 2, 3, 4]
```

So many side effects have been caused! Functions like these make reasoning about program behaviour incredibly difficult. Converting this function into a pure one (removing all side-effects) makes its behaviour clearer and more transparent.

```python
def f(ls):
  x = ls
  addend = x[-1] + 1
  x = x + [addend]
  ls = x + [addend + 1]
  return ls

ls = [1, 2, 3]
x = ls

print(f(ls)) # [1, 2, 3, 4, 5]
print(ls) # [1, 2, 3]
print(x) # [1, 2, 3]
```

Recursion

You have seen this before—use *recursive* functions to simulate loops.[3] Let's look at an example of a perfectly reasonable way to sum the numbers of a 2-dimensional list, using the `sum2D` function:

```
def sum2D(ls):
  total = 0
  for row in ls:
    for num in row:
      total += num
  return total
```

Loops are typically useful for its side-effects, primarily mutation. Looking at the (nested) loop above, a bunch of mutation occurs: the reassignments to `row` and `num` (the loop variables), and the mutation of the `total` variable in the loop body. In an environment where mutation is impossible, can we write the same program? Yes! Like we have said, rely on **recursion**! An example recursive formulation of the `sum2D` function from above would be like so:

```
def row_sum(row):
    return 0 if not row else \
           row[0] + row_sum(row[1:])

def sum2D(ls):
    return 0 if not ls else \
           row_sum(ls[0]) + sum2D(ls[1:])
```

Again, the behaviour of the program has not changed: the `sum2D` function still produces the correct output given any 2-dimensional list of integers. However, our function is still pure and does not mutate **any** data structure or variable.

Recursive solutions can also be more elegant, especially when the problem or data structures used are (inherently) recursive. Take the example of obtaining the preorder of a binary tree. Binary trees are recursive data structures, if formulated the following way:

A (nonempty) binary tree is either:

- A node with a value, a left tree and a right tree; OR
- A leaf with just a value

As you can see, the definition of a node contains (sub)trees, making the binary tree a recursive data structure[4]. Therefore, operations on trees can often be expressed elegantly using recursion. For example, the specification of obtaining the *preorder* of a tree can be like so:

1. The preorder of a leaf is a list containing the leaf's value
2. The preorder of a node is the node's value, together with the preorder of the left (sub)tree, then the preorder of the right (sub)tree.

This specification written in code is concise and elegant:

```python
from dataclasses import dataclass

@dataclass
class Tree: pass

@dataclass
class Node(Tree):
    val: object
    left: Tree
    right: Tree

@dataclass
class Leaf(Tree):
    val: object

def preorder(tree):
    match tree:
        case Node(val=v, left=l, right=r):
            return [v] + preorder(l) + preorder(r)
        case Leaf(val=v):
            return [v]
```

Recursive functions are also amenable to **formal** reasoning. Some languages (usually *Interactive Theorem Provers*) support proofs and can even automatically synthesize proofs of correctness for you. In the following example written in Lean 4, the following program defines a binary tree and a program for obtaining the preorder of the tree just as before; the key difference being, that Lean automatically helps us prove that the function **terminates**. In such an environment, we rarely have to worry whether our program gets stuck or crashes.

```
inductive Tree (α : Type) : Type where
  | node : α -> Tree α -> Tree α -> Tree α
  | leaf : α -> Tree α

-- compiler automatically synthesizes proof of termination
def Tree.preorder { β : Type } : Tree β -> List β
  | .node v l r => v :: (preorder l) ++ (preorder r)
  | .leaf v => [v]

def myTree : Tree Nat := .node 1 (.leaf 2) (.leaf 3)
#eval myTree.preorder -- [1, 2, 3]
```

The primary reason for this is that recursive functions can often be reasoned about via *induction*:

$$\frac{P(0) \qquad \forall k \in \mathbb{N}. P(k) \rightarrow P(k+1)}{\forall n \in \mathbb{N}. P(n)} \text{Induction}$$

We have seen that factorial can be written recursively, and in fact we can prove its correctness (in a quite straightforward manner) via induction. This makes the following factorial function implementation obviously correct.

```
-- Lean 4
def fac : Nat -> Nat
  | 0     => 1
  | n + 1 => (n + 1) * fac n
```

Types

Adhering strictly to type information **eliminates type-related bugs** and makes functions **transparent**. Perhaps most importantly, adherence to type information can be verified by a program.

Observe the following program fragment.

```
x: int = 123
# ...
print(x + 5)
```

If we fix the type of `x` to `int` and strictly adhere to it, then the last line containing `x + 5` will definitely not cause a `TypeError`, because we know that adding any number to an integer will always work.

Contrast the above with the following example.

```
# Python
def safe_div(num: int, den: int) -> int:
    return None if den == 0 else \
           num // den

x = int(input())
y = int(input())
z = safe_div(x, y) + 1 # hmmm...
print(z)
```

If we do not adhere to typing information strictly, no one knows that the `safe_div` function could return `None`! In such a scenario, if the user enters `0` for `y`, the expression `safe_div(x, y) + 1` would give a `TypeError`!

Function purity and adhering to types forces functions to be **transparent in effects**. That is because if we want our pure function to perform some effectful computation (such as potentially returning `None`), we must return an object that encapsulates this behaviour;

coupled with adhering to types, we must assign the correct type for the output of the function—the type of the object which encapsulates this behaviour—making the function's effects obvious.

To improve the program written earlier, let us try to create a data structure `Maybe` that is one of two things: `Just` a value, or `Nothing`. We can express this as dataclasses in Python (you may ignore the stuff involving `typing` and all the square brackets for now, they will make sense later).

```
from typing import Any
from dataclasses import dataclass

@dataclass(frozen=True)
class Maybe[T]:
    """Represents computation that may result in nothing"""
    pass

@dataclass(frozen=True)
class Just[T](Maybe[T]):
    j: T

@dataclass(frozen=True)
class Nothing(Maybe[Any]):
    pass
```

Now we can amend our `safe_div` function appropriately to return a `Maybe` value:

```
def safe_div(num: int, den: int) -> Maybe[int]:
    return Nothing() if den == 0 else \
           Just(num // den)
```

Notice two things: 1) the function is pure, and does nothing other than receive inputs and returns output 2) the function's type signature makes it incredibly obvious that the function will *maybe* produce an `int`. Therefore, users of this function are *forced* to handle the case where the function produces `Nothing`.

From this, we may proceed to use the `safe_div` function as before, except that instead of directly assigning `z = safe_div(x, y) + 1`, we must first call `safe_div` and handle the two cases: one where some integer was returned, the other where nothing was.

```
x: int = int(input())
y: int = int(input())
z: Maybe[int]
match safe_div(x, y):
    case Just(j):
        z = Just(j + 1)
    case Nothing():
        z = Nothing()
```

Types and type systems are highly useful, not just for verification of type safety, but also more generally, program verification and theorem proving etc. Types are backed by a rich theory (type theory) and is widely studied. As an example, interactive theorem provers may rely on systems with advanced type systems (such as the calculus of constructions, which has *dependent types*) to form the computational basis for proof assistance and proof checking. When these systems are baked into the language, we can write proof-carrying code and theorems (mathematical theorems or theorems about properties of code itself). An example is as follows, where theorems about the additive identity and the commutativity of addition of numbers can be used to show that concatenating a vector (like an immutable list) of length n to one of length k gives a vector of length $n + k$.

```
-- Lean 4
theorem izero : ∀ (k : Nat) , k = 0 + k
  | 0 => by rfl
  | n + 1 => congrArg (. + 1) (izero n)

theorem isucc (n k : Nat) : n + k + 1 = n + 1 + k :=
  match k with
  | 0 => by rfl
  | x + 1 => congrArg (. + 1) (isucc n x)

def Vect.concat {α : Type} {n k : Nat} : Vect α n -> Vect α k -> Vect α (n + k)
  | .nil, ys => izero k ▸ ys
  | .cons x xs, ys => isucc _ _ ▸ .cons x (xs.concat ys)
```

First-Class Functions

You might have seen in IT5001 that in some languages, functions are *first-class* objects.[5] This gives rise to higher-order functions which support **code re-use**. *Higher-order functions* can receive functions as arguments and/or return functions as output.

In the following program fragment, the `map` method of `Tree`s receive a function and returns a new tree with the function applied to all of its values. We then also *curry* the `add` function so that it receives the first addend, then returns a function that receives the second addend and returns the sum. This way, adding 2 to the values of a tree is as simple as several function calls:

```python
@dataclass(frozen=True)
class Tree:
    def map(self, f):
        match self:
            case Leaf(v):
                return Leaf(f(v))
            case Node(v, l, r):
                newval = f(v)
                newl = l.map(f)
                newr = r.map(f)
                return Node(newval, newl, newr)

@dataclass(frozen=True)
class Node(Tree):
    val: object
    left: Tree
    right: Tree

@dataclass(frozen=True)
class Leaf(Tree):
    val: object

def add(x):
    return lambda y: x + y

x = Node(1, Leaf(2), Leaf(3))
print(x.map(add(2))) # Node(3, Leaf(4), Leaf(5))
```

Functional programming languages emphasize this fact and make it easy and ergonomic to define higher-order functions. For example, in Haskell, functions are automatically curried, and has higher-order functions like `map` built into the standard library. This makes, for example, adding two to elements of a list, straightforward:

```haskell
main :: IO ()
main = do
  let x = [1, 2, 3]
  print (map (+2) x) -- [3, 4, 5]
```

So what?

Ideas from functional programming languages are increasingly being adopted in commonly-used imperative programming languages:

- Closures in C++/Rust/Java 8
- Structural pattern matching in Python 3.11/Java 21
- Algebraic Data Types in Rust
- Records in Java 14 etc.

Learning functional programming has a direct impact on your future work as a developer; functional programming is more than just a collection of language features and principles—it fundamentally encourages a new way of solving problem. As we've discussed, some of these principles impose meaningful constraints on programmers, which can make problem-solving more challenging and require innovative strategies. Nevertheless, mastering functional programming is invaluable, as it offers a fresh perspective on problem-solving. The skills you acquire will not only enhance your discipline as a developer but also empower you to explore diverse approaches to the challenges you encounter in your daily work.

Our goal for this course is to therefore first learn how to write programs in a purely functional programming language (thus forcing you to write programs fully with FP), and then transfer concepts into commonly used programming languages. For this, we will be writing code in two languages: *Haskell* (a purely functional programming language) and Python (which you should all be relatively familiar with).

Things You Need

For this course, you will need the following software:

- The Glasgow Haskell Compiler (GHC) (recommended: GHC 9.4.8 or newer)
- Python 3.12 (note the version; we shall be using new features)
- Any text editor you like (Visual Studio Code, Neovim etc.)

---

[1] Polymorphism in OOP refers to *subtype polymorphism*, which is different to the polymorphism in FP known as *parametric polymorphism*.

[2] If you have not, you may want to read a recap on the λ calculus before continuing.

[3] If you have not, you may want to read a recap on recursion before continuing.

[4] (Singly-linked) lists are also recursive data structures. To see this, look at our definition of binary trees, and remove one subtree in the definition of a node (therefore, a node has a value and one subtree). This is now a singly-linked list.

[5] If you have not, you may want to read a a recap on first-class functions before continuing.

LAST UPDATED **26 OCT 2024**

Haskell

Haskell is a *statically-typed*, *purely functional nonstrict-evaluation* programming language. Informally, static typing means that we can look at a program (without executing it) and tell what the type of any term is. A purely-functional language is a language that supports only functional programming concepts (unlike multi-paradigm languages like Python). Nonstrict-evaluation means that there is no strict sequence of evaluating statements or expressions, and compilers are free to decide which expressions should be evaluated first—*lazy evaluation* is where expressions are evaluated only when they are needed. We will look at non-strict evaluation eventually; for now, understanding static typing and purely functional programming is more important.

In a purely functional language like Haskell, you will miss the following programming language features that are present in virtually every general-purpose programming language:

- Mutation (even variables are immutable);
- Loops;
- Objects (classes etc.);
- Dynamic typing (e.g. `x` can be an `int` now, and a `str` later);

You might find it difficult to adjust to such a programming environment. However, you will find these restrictions meaningful as we have alluded to in the previous section.

Basic Expressions

By this point you should have already installed GHC, which comes with two main parts: `ghc` itself (the compiler), and `ghci` the REPL/interpreter. For now, run `ghci` in the terminal to start an interactive Haskell shell, and enter some basic mathematical expressions!

```
ghci> 1 + 2 - 3
0
ghci> 1 * 2 / 4
0.5
ghci> 5 ^ 2 `mod` 5
0
ghci> 5 `div` 2
2
```

Note some differences: `^` is exponentiation (just as you would normally type in a calculator), and there is no modulo operator. There is a modulo function called `mod`, and you can apply any binary function in an *infix* manner by surrounding the function in backticks. Integer division is a function `div`. The operator precedence rules apply.

In a functional programming language like Haskell, it should come as no surprise that virtually everything is a function. Mathematical operators are actually just functions! In GHCI, we can observe the type of any term (terms are sort of like objects in Python; functions are terms!) using `:t`, and we can show the type of the function of the `+` operator by issuing `:t (+)` (when writing operators as a term in the usual prefix notation, surround it in parentheses). We can in fact re-write an infix operator function call as a normal prefix function call. Note that in Haskell, `f x y z` is essentially the same as `f(x, y, z)` in languages like Python.

```
ghci> :t (+)
Num a => a -> a -> a
ghci> 2 + 3
5
ghci> (+) 2 3
5
```

As we know, currying is the act of translating an n-ary function to a unary function that receives one parameter and returns a function that receives the remaining parameters (in curried form). In Haskell, all functions are curried, so even a function like `(+)` really looks something like this in Python:

```
def add(x):
    return lambda y: x + y
```

This is automatically done in Haskell. Thus we might be able to write our Python equivalent of `add(2)` directly in Haskell as `(+2)`:

```
ghci> y = (+2)
ghci> y 3
5
```

which in Python, looks like:

```
>>> def add(x): return lambda y: x + y
>>> y = add(2)
>>> y(3)
5
```

Therefore, to be more specific, `f x y z` in Haskell is more like `f(x)(y)(z)` in Python.

We can also load Haskell source files into GHCI. Python source files have the `.py` extension; Haskell source files instead have the `.hs` extension. Let us try writing a simple Haskell

program. Create a new file like `MyCode.hs` and write in the following:

```haskell
-- MyCode.hs
main :: IO () -- entry point to the program
main = putStrLn "Hello World!"
```

We will look at what the first line means in the future. For now, try compiling and running your code by issuing the following commands in your terminal (windows users might have to run `./MyCode.exe`):

```
ghc MyCode.hs
./MyCode
```

The first command invokes GHC to *compile* your source file. *Compilation* translates your source file into an *executable* file that your computer that understand. The compilation process will also perform a bunch of compile-time checks, such as type-checking etc. It may also perform some optimizations. The outcome of invoking that command is an executable (probably called `MyCode`) along with other files (which we shall not talk about for now). The second command then executes that executable, and you should see `Hello World!` shown in the terminal.

```
Hello World!
```

We shall ignore compiling source files for now and temporarily focus on working with GHCI. In GHCI, we can load files by issuing `:l MyFile.hs`, which loads the source into the shell. For now, write the following code in `MyCode.hs`:

```haskell
-- MyCode.hs
z = 1 -- ok
y = 2 -- ok
y = 3 -- not ok!
```

As we have described earlier, everything in Haskell is immutable. Therefore, re-defining what `y` is should be disallowed! Let's try loading `MyCode.hs` into GHCI:

```
ghci> :l MyCode.hs
[1 of 2] Compiling Main ( MyCode.hs, interpreted )

MyCode.hs:4:1: error:
    Multiple declarations of 'y'
    Declared at: MyCode.hs:3:1
                 MyCode.hs:4:1
  |
4 | y = 3 -- not ok!
  | ^
```

As you can see, you cannot redefine functions or variables. Everything is immutable in Haskell! Therefore, the statement `x = e` is **not** an assignment statement. Rather, it is a *bind*

or a *definition*.

Control Structures

In Haskell, you mainly write *expressions*, and not statements. Consequently, there are only `if - else` expressions, and no `if - else` statements. That means that you cannot omit an `else` branch of an `if - else` expression, just like in Python:

```
>>> x = 2 * -1
>>> y = 'positive' if x == 2 else 'negative'
>>> y
'negative'
```

In Haskell, this would be (negative numbers must be surrounded by parentheses, otherwise Haskell thinks it is a partial function application of subtraction `(-)`):

```
ghci> x = 2 * (-1)
ghci> y = if x == 2 then "positive" else "negative"
ghci> y
"negative"
```

Just like in Python, `if - then - else` expressions in Haskell are *expressions* and therefore evaluate to a term:

```
ghci> (if 1 /= 2 then 3 else 4) + 5
8
```

Note that *not equals* looks like `/=` in Haskell but `!=` in Python. The equivalent expression in Python might be:

```
>>> (3 if 1 != 2 else 4) + 5
8
```

Importantly, the type of any expression is fixed, or at least, we should be able to determine what the type of every expression is unambiguously just by looking at it. Therefore, writing the following expression in Haskell will throw an error:

```
ghci> x = 2 * (-1)
ghci> y = if x == 2 then 2 else "negative"
<interactive>:2:20: error:
  - No instance for (Num String) arising from the literal '2'
  - In the expression: 2
    In the expression: if x == 2 then 2 else "negative"
    In an equation for 'y': y = if x == 2 then 2 else "negative"
```

The reason is that we should not need to evaluate the truth of `x == 2` to determine what the type of the entire `if - else` expression is. Thus, Haskell requires that the type of the expression in the `if` branch be the same as the type of the expression in the `else` branch. This departs from Python which is *dynamically typed*, where types are determined at runtime, so expressions can freely be of different types based on the values they inherit at the time of program execution.

Functions

Defining functions in Haskell looks like defining a variable. This should be expected since Haskell is centred around functions, so it should come as no surprise that functions do not need to be defined with any special syntax.

```
ghci> oddOrEven x = if even x then "even" else "odd"
ghci> oddOrEven 1
"odd"
ghci> oddOrEven 2
"even"

ghci> quadratic c2 c1 c0 x = c2 * x ^ 2 + c1 * x + c0
ghci> f = quadratic 1 2 3 -- x^2 + 2x + 3
ghci> f 4
27
ghci> f 5
38
```

We might then ask: how do we write a loop in Haskell? Like we said earlier, Haskell is a purely functional programming language, so there are no loops (we may later see loops being simulated with functions). Thus, for now we shall use recursion as it is often the most elegant way to solve problems.

Recall that the familiar `factorial` function may be written imperatively in Python as:

```
def fac(n):
    res = 1
    for i in range(2, n + 1):
        res *= i
    return res
```

As we know, the factorial function can be defined recursively as such:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

And in Python:

```
def fac(n):
    return 1 if n == 0 else \
           n * fac(n - 1)
```

In Haskell, we are free to do the same:

```
ghci> fac n = if n == 0 then 1 else n * fac (n - 1)
ghci> fac 4
24
```

In fact, we can also express functions like this elegantly in Haskell with *guards*. Guards allow us to define expressions differently based on a condition.

For example, we know that the Fibonacci function may be written like so:

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

And writing this function with regular `if - else` expressions might look like: [1]

```
ghci> :{
ghci| fib n = if n == 0 || n == 1
ghci|         then 1
ghci|         else fib (n - 1) + fib (n - 2)
ghci| :}
```

However, it might look clearer to define it this way with guards (`otherwise` is just defined as `True`):

```
ghci> :{
ghci| fib n
ghci|   | n == 0     = 1
ghci|   | n == 1     = 1
ghci|   | otherwise = fib (n - 1) + fib (n - 2)
ghci| :}
ghci> fib 5
8
```

Even better, we can use *pattern matching* to define such functions much more easily. We will look at pattern matching in more detail in the future:

```
ghci> fib 0 = 1
ghci> fib 1 = 1
ghci> fib n = fib (n - 1) + fib (n - 2)
ghci> fib 5
8
```

Auxiliary Bindings

Thus far we have defined functions as a single expression; this is akin to writing a *lambda expression* in Python. As we know, that may not always be the most ergonomic considering that many functions can be better defined with several 'statements' that lead into a final expression. One example would be the following in Python:

```python
def weight_sum(n1, w1, n2, w2):
    x = n1 * w1
    y = n2 * w2
    return x + y
```

While it is completely acceptable to define this function in one line, it is not as readable. In Haskell, functions indeed have to be written as a single expression, but we can define local bindings for the expression using `let`:

```
ghci> :{
ghci| weightSum n1 w1 n2 w2 =
ghci|   let x = n1 * w1
ghci|       y = n2 * w2
ghci|   in  x + y
ghci| :}
ghci> weightSum 2 3 4 5
26
```

The `let` binding allows us to introduce the definitions of `x` and `y` which are used in the expression after the `in` clause. These make writing larger expressions more readable.

`let` bindings are (more-or-less) *syntax sugar* for function calls:

```
weightSum n1 w1 n2 w2 =
    let x = n1 * w1
        y = n2 * w2
    in  x + y

-- same as

weightSum n1 w1 n2 w2 =
    f (n1 * w1) (n2 * w2)

f x y = x + y
```

Importantly, `let` bindings are expressions; they therefore evaluate to a value, as seen in this example:

```
ghci> (let x = 1 + 2 in x * 3) + 4
13
```

This is different to `where` bindings, which also allow us to write auxiliary definitions that support the main definition:

```
weightSum n1 w1 n2 w2 =
    let x = n1 * w1
        y = n2 * w2
    in  x + y

-- same as

weightSum n1 w1 n2 w2 = x + y
    where x = n1 * w1
          y = n2 * w2
```

Other differences between `let` and `where` are not so apparent at this stage. You are free to use either appropriately (use `let` where an expression is desired, using either `let` or `where` are both okay in other scenarios).

Data Types

We have looked at some simple data types so far: numbers like `1.2`, and strings like `"abc"`. Strings are actually **lists** of characters! Strings are surrounded by double quotes, and characters are surrounded by single quotes, like `'a'`.

Lists in Haskell are *singly-linked list* with homogenous data. That means that the types of the elements in the list must be the same. We can write lists using very familiar syntax, e.g. `[1, 2, 3]` being a list containing the numbers 1, 2 and 3. Indexing a list can be done with the `!!` function.

```
ghci> x = [1, 2, 3]
ghci> x !! 1 -- indexing, like x[1]
2
```

We can also construct ranges of numbers, or any enumerable type (such as characters). The syntax for creating such lists is straightforward as shown in the examples below.

```
ghci> y = [1,3..7] -- list(range(1, 8, 2))
ghci> y
[1,3,5,7]
ghci> z = [1..10]  -- list(range(1, 11))
ghci> z
[1,2,3,4,5,6,7,8,9,10]
ghci> inflist = [1..] -- 1,2,3,...
ghci> inflist !! 10
11
```

As we stated earlier, strings are lists of characters, we can even build ranges of characters which result in strings.

```
ghci> ['h', 'e', 'l', 'l', 'o']
"hello"
ghci> ['a'..'e']
"abcde"
ghci> ['a'..'e'] ++ ['A'..'D'] -- ++ is concatentation
"abcdeABCD"
```

As you know, a singly-linked list is one of two things: an empty list, or a node with a value (`head`) and a reference to the remaining part of the list (`tail`). Thus, one of the most frequently used operations is the *cons* operation (`:`) which builds (or de-structures) a list given its head and tail values. The `:` operator is right-associative.

```
ghci> x = [1, 2, 3]
ghci> 0 : x
[0,1,2,3]
ghci> 0 : 1 : 2 : 3 : []
[0,1,2,3]
ghci> 'a' : "bcde"
"abcde"
```

One of the most interesting parts of Haskell is that it has non-strict evaluation. That means that the compiler is free to evaluate any expression only when it is needed. This allows us to quite nicely define recursive data without running into infinite loops:

```
ghci> y = 1 : y
ghci> take 5 y
[1,1,1,1,1]
```

As we know, performing recursion over a list frequently requires us to get a head element and then recursively calling the function over the remaining list. This is nicely supported without any performance costs unlike in Python, where `ls[1:]` runs in $\mathrm{O}(\mathrm{n})$. For example, writing a function that sums a list of numbers might look like the following in Python:

```
def sum(ls):
    if len(ls) == 0:
        return 0
    return ls[0] + sum(ls[1:])
```

Haskell is very similar (`head` is a function that returns the first element of a list, and `tail` is a function that returns the remainder of a list):

```
sum' ls = if length ls == 0
          then 0
          else head ls + sum' (tail ls)
```

As a quick aside, the `:` operator is really a *constructor* for lists, so in fact we can use pattern matching (again, we will discuss this in the future) to define the `sum'` function very elegantly.

```
sum' [] = 0
sum' (x : xs) = x + sum' xs
```

Python also supports *list comprehension* as you may recall:

```
>>> x = [1, 2, 3]
>>> y = 'abc'
>>> [(i, j) for i in x for j in y if i % 2 == 1]
[(1, 'a'), (1, 'b'), (1, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
```

Haskell also provides the same facility, with different syntax:

```
ghci> x = [1, 2, 3]
ghci> y = "abc"
ghci> [(i, j) | i <- x, j <- y, odd i]
[(1,'a'),(1,'b'),(1,'c'),(3,'a'),(3,'b'),(3,'c')]
```

At this junction it would be most appropriate to discuss tuples. Like Python, the fields of a tuple can be of different types. However, tuples in Haskell are **not** sequences. Tuples behave more like the product of several types, as is usually the case in many domains.

As such, there are not many operations we can do on tuples. One of the only special cases is pairs, which have functions to project each value:

```
ghci> fst (1,"abc")
1
ghci> snd (1,(2,[3,4,5]))
(2,[3,4,5])
ghci> snd (snd (1,(2,[3,4,5])))
[3,4,5]
```

This should suffice for now. Now is your turn to try the exercises to get you started on your functional programming journey! Note that many of the functions we have used are built-in to Haskell, as defined in Haskell's Prelude library. You may want to refer to this library when doing the exercises. A large portion of the Prelude documentation may be unreadable at this point, however, rest assured that many of the concepts presented in the documentation will be covered in this course.

[1] Note that `:{` and `:}` are used only in GHCI to define blocks of code, and are not part of Haskell.

LAST UPDATED **26 OCT 2024**

Exercises

Question 1

Without using GHCI, evaluate the results of the following expressions:

1. `3 * 4 + 5`
2. `3 + 4 * 5`
3. `5 ^ 3 `mod` 4`
4. `97 / 4`
5. `97 `div` 4`
6. `if (let x = 3 in x + 3) /= 5 && 3 < 4 then 1 else 2`
7. `not otherwise`
8. `fst (0, 1, 2)`
9. `succ (1 / 2)`
10. `sqrt 2`
11. `1 `elem` [1, 2, 3]`
12. `let f x = x + 1; g x = x * 2 in (g . f) 1`
13. `[1, 2, 3] ++ [4, 5, 6]`
14. `head [1, 2, 3]`
15. `tail [1, 2, 3]`
16. `init [1, 2, 3]`
17. `[1, 2, 3] !! 0`
18. `null []`
19. `length [1, 2, 3]`
20. `drop 2 [1, 2, 3]`
21. `take 5 [-1..]`
22. `dropWhile even [2, 6, 4, 5, 1, 2, 3]`
23. `sum [fst x | x <- [(i, j) | i <- [1..4], j <- [-1..1]]]`

Question 2

Write a function `eqLast` that receives two nonempty lists and checks whether the last element of both are the same. Example runs follow:

```
ghci> eqLast [1,2,3] [4,5]
False
ghci> eqLast "ac" "dc"
True
```

Question 3

A palindrome is a word that reads the same forward or backward. Write a function `isPalindrome` that checks if a string is a palindrome. Example runs follow:

```
ghci> isPalindrome "a"
True
ghci> isPalindrome "bcde"
False
ghci> isPalindrome "racecar"
True
```

Question 4

You are writing a function to determine the cost of a ride. The cost of a ride is determined by $f + rd$ where f is the flag down fare, r is the per km rate of the ride and d is the distance of the ride in km. Write a function `taxiFare` that receives f, r and d and computes the total cost. Example runs follow:

```
ghci> grab = taxiFare 3 0.5
ghci> gojek = taxiFare 2.5 0.6
ghci> grab 3
4.5
ghci> gojek 3
4.3
ghci> grab 10
8.0
ghci> gojek 10
8.5
```

Question 5

Nowadays, we can customize the food that we order. For example, you can order your burger with extra or no cheese. In this exercise, we will write a function that takes a string as the customization and compute the price for burgers with the code names for the customization. You are given the price list for ingredients:

Ingredient	Price
`B` for bun	$0.50
`C` for cheese	$0.80
`P` for patty	$1.50
`V` for veggies	$0.70
`O` for onions	$0.40
`M` for mushrooms	$0.90

Write a function `burgerPrice` that takes in a burger as a string of characters (each character represents an ingredient in the burger) and returns the price of the burger. While doing so, define an auxilliary function `ingredientPrice` that receives a single ingredient (as a character) and returns its price. Define `ingredientPrice` as part of `burgerPrice` using a `where` binding. Example runs follow:

```
ghci> burgerPrice "BVPB"
3.2
ghci> burgerPrice "BVPCOMB"
5.3
```

Question 6

Write a function `sumDigits` that receives a nonnegative integer and gives the sum of its digits. Example runs follow:

```
ghci> sumDigits 123
6
ghci> sumDigits 12356
17
```

Question 7

Write a function `@:` that receives a list and a tuple of two values `(start, stop)`, and performs list slicing with indices starting from `start` and ending at (and excluding) `stop`. The step size is 1. Assume that both the start and stop values are nonnegative integers. Example runs follow:

```
ghci> [1, 2, 3] @: (1, 4)
[2,3]
ghci> [1, 2, 3] @: (4, 1)
[]
ghci> [1, 2, 3] @: (0,1)
[1]
ghci> [1, 2, 3] @: (1,67)
[2,3]
```

Syntactically, the way to define this function might be the following:

```
ls @: (start, stop) = your implementation here
```

LAST UPDATED **26 OCT 2024**

As per the course title, one of the most important aspects of functional programming that we shall cover is *types*. In this chapter, we shall describe what types are, how they are useful and how we can use type information to write code, and some aspects of types that allow us to reduce boilerplate code while still retaining type-safety. In addition, we describe how we can define our own data types in Haskell, and a neat feature known as *pattern matching* that is extremely useful in the presence of *algebraic data types*. We also offer some examples of how we can incorporate these concepts in Python.

LAST UPDATED **26 OCT 2024**

Type Systems

As the course title suggests, Haskell is a typed functional programming language—in particular, it uses a statically-typed *type system*. This begs the question, "what is a type system?"

An online search for definitions might give you the following:

> **Definition (Type System)**. A type system is a **tractable syntactic method** for **proving the absence of certain program behaviours** by classifying phrases according to the **kinds of values they compute**.

Let us unpack the highlighted phrases in the definition above.

Tractable syntactic method

Tractable more or less means *easy*, or *polynomial time*. *Method* refers to a *formal method*, which means it is a kind of mathematically formal process. The fact that it is a *syntactic method* means that this formal analysis can be done syntactically, without the need to appeal to a *semantic* analysis (although, static type checking is done against the static semantics of the type system). More or less, it can be performed without executing any of the code it is analyzing.

Proving the absence of certain program behaviours

In the case of type systems, this usually means that the type system is used to prove the absence of type errors. The realm of program analysis is broken down into roughly two kinds: over-approximation analyses, and under-approximation analyses. Notice that both perform *approximations* of program behaviour—this is because obtaining a precise specification of any program is *undecidable*. Generally, static analyses, like type checking, perform an over-approximation of program behaviour. An analogy of how this works is as follows: assume true program behaviour is x and buggy behaviour is at y (these are all positive numbers, let's say). We then over-approximate the true program behaviour, giving us $\mathrm{x} + \epsilon$. If we can show that $\mathrm{x} + \epsilon < \mathrm{y}$, then we can guarantee that $\mathrm{x} < \mathrm{y}$, so the program is not buggy.

A more concrete example is as follows. Let's suppose we have the following code snippet in Python:

```
y: int = 0 if f() else 'abc'
print(y + 1)
```

Notice that if we can determine that `f` always returns `True`, then we know for sure that there will be no type errors. However, it is not possible to make this determination in general. Thus, we over-approximate program behaviour by assuming that it is possible that `f` may return either `True` or `False` leading us to show that we cannot prove the absence of type errors in this program. Instead, if we had written the following:

```
y: int = 0 if f() else 1
print(y + 1)
```

Then even by assuming that both branches of the conditional expression may be the result, we can conclusively show that `y` will always be an `int`. Our over-approximation of program behaviour doesn't have type errors, meaning, that our actual program really does not have type errors.

Kinds of values they compute

This is a simple description of what *types* are. Types, as we will informally define later, are classifications of data/things in the program that all behave similarly or have similar characteristics. In some other sense, types can be seen as abstractions over terms.

Simply put, a type system is a formal system that lets us show that there won't be type errors. As we have seen, the nature of [statically-typed] type systems forces us to program in a different way (at least compared to dynamically typed languages like Python), and this is what we will explore in this chapter.

Types

Type systems are systems of types; but *what is a type*? In essence, a type is like a *kind* of thing, or a high-level description of what something is. Types (1) give meaning to some data, and (2) describe what its members are like.

Since you have already programmed in Python, you should have some inkling of what types are. In Python, everything is an object. Thus, in Python, the type of an object is the *class* from which it was instantiated.

The following is some sample output showing the types of various objects. The output of all these function calls are classes.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> type('abc')
<class 'str'>
>>> class A: pass
>>> type(A())
<class '__main__.A'>
```

This is very apt—classes are blueprints for creating objects, and (for the most part), all *instances* of a class will abide by the specification as laid out in the class. Therefore, Python's type system based on classes is very appropriate for our purposes. In fact, this is not unique to Python. Many other languages with OO features also have classes as types.

In Python, we mainly think of types as being bound to *objects*, that is, objects have *reified* types that can be accessed at runtime. We have never thought of assigning types to variables or function parameters, since when we are investigating the type of a variable, what we are really doing is investigating the type of the object that is referred to by the variable. However, Python actually does allow us to annotate variables, function parameters etc with types to document "suggestions" as to what the types of the objects assigned to them should be.

Observe the following program fragment.

```
def f(x: int) -> str:
    y: int = x * 2
    return f'{x} * 2 = {y}'
z: int
z = 3
s: str = f(z)
print(s) # 3 * 2 = 6
```

This program fragment contains several *type annotations*. In the function header, we have a specification for `f` to receive an `int` and return a `str`. That is, if the type annotations make sense, then passing an `int` into `f` will always result in a `str`. In the function body, we also have an annotation for the variable `y` stating that it is also an `int`. This makes sense—if `x` is an `int`, then so will `x * 2`. Actually, the type of `y` can be *inferred* (a type checker can determine the type of `y` automatically), so our type annotation for it is not necessary. Outside the function body we have other type annotations, documenting what the types of the other variables are. On visual inspection, we can see that all the type annotations make sense and we have adhered to them fully; we are thus guaranteed that we have no type errors.

While Haskell also provides the capability for type annotations, a notable distinction lies in Haskell's *enforcement* of adherence to these annotations. Consequently, it might be more fitting to refer to them as *type declarations*. Nevertheless, the core concept remains unchanged: specifying the types of variables, functions, or terms ensures that, when adhered to correctly, our program will be well-typed.

The following code snippet shows some Haskell code with type declarations.

```haskell
f :: Int -> String
f x = show x ++ " * 2 = " ++ show y
    where y = x * 2
z :: Int
z = 3
s :: String
s = f(z) -- 3 * 2 = 6
```

A natural question would be to ask, what types can we declare variables to be of? We have looked at some basic types earlier, `Int`, `String` (which is an alias for `[Char]`), `Char`, `[Int]`, `Bool`, `Double` etc. There are many other types in Haskell's Prelude, and later on we will see how we can create our own types.

Declaring types for functions is slightly different. In Python, when writing type annotations for functions, we are really annotating the types of its parameters, and its return type. In Haskell, we are declaring the type of the function itself. The difference is actually not as large as one might imagine. If the function receives a type S and returns a type T, then the function has the type $\mathrm{S} \to \mathrm{T}$. We similarly use arrows to declare the type of functions in Haskell. Thus, as above, since `f` receives an `Int` and returns a `String`, then `f` itself is of the type `Int -> String`.

Haskell has roots in formal systems, in particular, System $\mathrm{F_C}$, which is a dialect of System $\mathrm{F}\omega$ (without type lambdas). Thus, the types of terms can be described formally. Knowing the formal typing rules of Haskell is not required, but may give you some insight as to how it works. Below we show the typing rules for function declarations, more accurately, lambda abstractions.

$$\frac{\Gamma, \mathrm{x} : \mathrm{S} \vdash \mathrm{e} : \mathrm{T}}{\Gamma \vdash \lambda \mathrm{x}.\, \mathrm{e} : \mathrm{S} \to \mathrm{T}}\text{T-Abs}$$

The T-Abs rule is an *inference rule* stating that if the premise above the line is true, then the conclusion below the line will also be true. Let's first parse the premise. The part to the left of $\vdash$ is the *typing environment*, more or less describing the type declarations we have at the point of analysis of the program. Specifically, Γ is the actual type environment, while $\mathrm{x} : \mathrm{S}$ is an additional assumption that a variable x has type S. The part to the right of $\vdash$ describes the judgement of the type of e being T. Overall, the premise states "given what we have so far, if in assuming x is of type S we get that e is of type T, ...". The conclusion can be understood similarly: it states that the typing environment Γ will show that the function $\lambda \mathrm{x}.\, \mathrm{e}$ has type $\mathrm{S} \to \mathrm{T}$. Putting these together, the rule states that "given typing environment Γ, if by assuming that variable x has type S we get that the expression e is of type T, then Γ will also show that the type of the function $\lambda \mathrm{x}.\, \mathrm{e}$ is of type $\mathrm{S} \to \mathrm{T}$".

A simple demonstration in Python is as follows: suppose we have x as `x` and e as `x * 2`. If we assume that `x` is of type `int`, then we know that `x * 2` will also be an `int`. Therefore, the type of $\lambda \mathrm{x}.\, \mathrm{e}$ which is `lambda x: x * 2` is `int -> int` [1].

What about multi-parameter functions? Remember that in Haskell, all functions are curried, thus, all functions in Haskell are single parameter functions. Curried functions receive one parameter, and return a function *closure* that receives the remaining variables and eventually will return the final result. Therefore the `(+)` function actually looks more like:

```python
# Python
def add(x):
    return lambda y: x + y
```

The type of `add` is more like `int -> (int -> int)`. This is (more or less) the type of `(+)` in Haskell, which (more or less) has type `Int -> Int -> Int`. Note that `->` is right-associative, so `Int -> Int -> Int` is the same as `Int -> (Int -> Int)`.

In Haskell, the types of everything are **fixed**. This should be unsurprising since everything in Haskell is immutable, but it is a restriction that can also be found in other less restrictive languages like Java and C++. In this environment, we have to, perhaps ahead of time, decide what the type of a variable, function, function parameter is, then write the implementation of your function around those restrictions.

The following code snippet first *declares* the type of `f` before showing its implementation. It is not only good practice to declare types above their implementation, but it can be a nice way to frame your mind around the implementation of your function—start by providing a high-level specification of your function, then work on the implementation to describe what the function is actually trying to achieve.

```haskell
f :: Int -> String -- explicit type declaration
f x = show x ++ "!"
g x = x + 1 -- type of g is inferred
```

However, observe that the type of `g` is not defined. This does not mean that the type of `g` is dynamic or is not being checked; rather, Haskell can infer the *principal* (most liberal) type of `g` via a process known as *type inference*. That still means that the implementation of `g` itself must be well-typed (its implementation does not break any of the typing rules), and that any users of `g` must abide by its static type signature.

Generally speaking, it is good practice to declare the types of top-level bindings—that is, nested bindings of functions, variables (for example, in `let` expressions) do not need type declarations and can often be inferred. The example above of the declaration of `f` is a perfectly idiomatic way of defining and declaring a function, unlike `g` which lacks a type declaration.

Programming with Types

When learning Python, you might not have had to think very much about types; this is because Python does not care about type annotations. For example, you can happily

annotate a variable to be an `int` but then assign a string into it. This is very much unlike Haskell, where adherence to type declarations and well-typedness is *enforced* by the compiler—the compiler will reject any program that is not well-typed.

Observe the following program fragment:

```
f :: Int -> String -- explicit type declaration
f x = show x ++ "!"

g = f "1" -- compiler throws type error as f receives Int, not String
```

The definition of `f` is well-typed since it abides by all the typing rules, and all the types make sense. However, since `f` only receives `Int`, passing a `String` into it is a clear violation of the rules. Thus, the entire program is ill-typed and will not be compiled. Try this for yourself!

Programming in such a strict and formal language can feel restrictive, but these restrictions actually feel more like "guard rails" or "seatbelts"; if your program passes the checks done by the compiler, you can be quite assured that it works. As the saying goes, in Haskell, "if it compiles, it works". Although this is not necessarily true, Haskell's robust and expressive type system allows you to rule out a large class of bugs, and often, results in correct programs. However, one question to ask is: how do we go about programming with static types?

The first step of being able to program with types is understanding the typing rules. We shall elide explanation on how typing works with inference, typeclasses, polymorphism etc. and focus solely on the simplest typing rules:

1. In a binding `x = e`, the type of `x` must be the same as the type of `e`
2. In a conditional expression `if x then y else z`, the type of `x` must be `Bool` and the types of `y` and `z` must both be equal to some type `a`; the type of the entire expression is `a`
3. In a function application expression `f x` the type of `f` must be `a -> b` for some `a` and `b`, `x` must be of type `a`, and the type of the expression is `b`
4. (Without loss of generality of number of parameters) For a function binding `f x = e` the type of `f` must be `a -> b` for some `a` and `b` and in assuming `x` to be of type `a`, `e` must be of type `b`.

Try calculating the types of every expression in the following code snippet. Can you get it all right?

```haskell
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in  x : r
```

Let's work through this example.

- We are declaring `f` to be of type `Int -> Int -> [Int]`, so it stands to reason that in the definition of `f` we are assuming that `x` and `n` are both of type `Int`.
- For this to be well-typed, we must ensure that the conditional expression evaluates to `[Int]`, that means both branches must themselves evaluate to `[Int]`.
- First we observe the condition `n == 0`; the `(==)` function receives two numbers and returns a `Bool`, so this is well-typed.
- Looking at the `True` branch, we see that we are returning the empty list, which matches the type of `[Int]`.
- In the `False` branch, we have a `let` expression, so we must ensure that `x : r` evaluates to `[Int]` too.
- The `let` binding contains a binding `r = f x (n - 1)`; knowing that (by our own declaration) `f` has type `Int -> Int -> [Int]`, knowing that `x` and `n - 1` are of type `Int` means we can safely conclude that `r` has type `[Int]` (of course, the `(-)` function receives two integers and returns an integer).
- The `(:)` function receives an `Int` and a `[Int]` and returns a `[Int]`, so all the types match.

Overall, we have seen that we successfully determined the types of every expression in the program fragment, and concluded that it is well-typed.

Now that you are familiar with the basic typing rules and (roughly) how types are inferred, the next step is to get comfortable writing programs with static types. Generally this comes with practice, but one great way to get you started with typeful programming is to try letting the *types guide your programming*.

Suppose we are trying to define a function `f` that receives an integer `x` and returns a string showing the result of multiplying `x` by 2:

```
ghci> f 3
"3 * 2 = 6"
ghci> f 5
"5 * 2 = 10"
```

Let us try implementing this function. The first thing we have to consider is the type of `f` itself, which by definition, should receive an `Int` and return a `String`. As such, we may start with the type declaration `f :: Int -> String`.

Next, we know we are eventually going to have to convert `x` into a `String`. We know that there is a `show` function that does that. Its type signature (modified) is `Int -> String`, so we know that `show x` is a `String`.

We also know that we need to multiply `x` by 2. For this, we can use the `(*)` function, which has a (modified) type signature of `Int -> Int -> Int`. Thus, we can write `x * 2` and that gives us an `Int`. Knowing that we eventually need to display it as a `String`, once again, we can rely on the `show` function.

Now we have all the numbers we need in `String` form, we need to concatenate them together. For this, we can rely on our trusty `(++)` function that receives two `String`s and returns a `String`. Using this allows us to concatenate all our desired strings together. Since our original function `f` was meant to return a `String`, we can return it as our final result.

```
f :: Int -> String
f x =
  let sx :: String = show x
      y  :: Int    = x * 2
      sy :: String = show y
  in  sx ++ " * 2 = " ++ sy
```

This is a simple example of using types to guide your programming. While seemingly trivial, this skill can be incredibly useful for defining **recursive** functions!

Suppose we are trying to define a function that sums the integers in a list. As always, we must decide what the type of this function is. As per our definition, it receives a list of integers and returns the final sum, which should be an integer as well. This gives us the type declaration `sum' :: [Int] -> Int`.

First, let us define the base case. We should be quite clear on what the condition for the base case is: it should be when the input list is empty. What should we return in the base case? By our type declaration, we must return an `Int`, so we must express our base result in that type. The result is `0`, which matches our type declaration.

Next we must define the recursive case. This one might be tricky initially. We know that we can make our recursive call, passing in the tail of the input list. This might look something like `sum' (tail ls)`. We must be very clear about the type of this expression; as per the type declaration, the result is an `Int`, and not anything else.

We also know that we want to add the head of the input list to the result of the recursive call. In doing so we get an `Int`.

Finally, we can add the results together, giving us an `Int`, which matches our return type.

```
sum' :: [Int] -> Int
sum' ls =
    if null ls
    then 0
    else let r  :: Int = sum' (tail ls)
             hd :: Int = head ls
         in  hd + r
```

By getting used to types, having a statically-typed system no longer feels like a chore or a hurdle to cross, and instead feels like a support system that makes everything you are doing clear! Many developers (including myself) love statically-typed programming languages for this very reason, so much so that people have gone to great lengths to add static typing to otherwise dynamically typed languages like JavaScript (the typed variant of JavaScript is TypeScript).

Python is no different. Several static type checkers are out there to help us analyze the well-typedness of our program. One of the most popular analyzers is `mypy`, which was heavily developed by Dropbox. However, I recommend `pyright` because at the time of writing, it has implemented bleeding edge features that we need for further discussion of types which we shall see very shortly.

Let's see `pyright` in action. We shall write an ill-typed program and see if it catches the potential bug:

```
# main.py
def f(x: int, y: int) -> int:
    z = x / y
    return z
```

Running `pyright` on this program will reveal an error message:

```
pyright main.py
```

```
pyright main.py
/home/main.py
  /home/main.py:4:12 - error:
    Expression of type "float" is incompatible with return
    type "int"
    "float" is incompatible with "int" (reportReturnType)
1 error, 0 warnings, 0 informations
```

Great! This makes sense because assuming `x` and `y` are of type `int`, the type of `z` should actually be `float`! Let's correct the program and try running `pyright` against the new program:

```python
# main.py
def f(x: int, y: int) -> int:
    z = x // y
    return z


pyright main.py
0 errors, 0 warnings, 0 informations
```

Very well! We have now learnt how to program with types in Haskell and in Python, and since Python does not come with a type-checker, we are able to use tools like `pyright` to do the type checking for us!

One additional great feature about `pyright` is that it is actually also a language server. As such, you can include `pyright` in your favourite text editors so that it can catch bugs while writing programs!

---

[1] Python doesn't have arrow types. The actual type of the function is `Callable[[int], int]`.

LAST UPDATED **26 OCT 2024**

Polymorphism

In FP, functions describe computation and applying functions perform said computation. For example, given a function f:

$$\mathrm{f}(\mathrm{x}) = \mathrm{x} \times 2$$

f describes what computation is to be done (multiplying the parameter by 2), and applying f onto a value (such as $\mathrm{f}(2)$) performs the computation that gives the result, which is 4. Importantly, you might also find that applying it onto a different input may give you a different outcome. In this case, $\mathrm{f}(2) = 4 \neq \mathrm{f}(3) = 6$. The output depends on the input, i.e. we have *terms that depend on terms*. This may at first glance seem like a trivial observation because that is what functions are designed to do: if functions are always constant like $\mathrm{g}(\mathrm{x}) = 1$ then we can always replace all applications of the function with the result and no computation needs to be done.

However, now that we have learnt about types, we get a much more interesting avenue for extending this idea of dependence. In fact, we now have three orthogonal directions to explore[1]:

1. Can terms depend on types?
2. Can types depend on types?
3. Can types depend on terms?

The answer to the first two questions is yes! This phenomenon is known as [parametric] *polymorphism*, i.e. where types and terms can depend on types[2].

Polymorphic Types

Let us motivate this need with an example. Suppose we are trying to create a wrapper class called `Box`, that contains a single value. As per usual, we have to think about the type of the value it contains. At this point we cannot simply allow the value to be *anything*, so we shall fix the type of the value to something, say, `int`.

```python
# Python
@dataclass
class IntBox:
    value: int
```

However, we may later want a `Box` that stores strings. In this case, we will have to define a new class that does so.

```
# Python
@dataclass
class StrBox:
    value: str
```

Recall one of the core principles in programming: whenever you see a pattern in your code, *retain similarities* and *parameterize* differences. Looking at the two `Box` implementations, you should be able to see that the implementation is virtually identical, and the only difference is the *type* of `value`. We have previously been able to parameterize values (regular function parameters), parameterize behaviour (higher-order functions), however, can we parameterize *types*?

Yes! We can define `Box` to receive a *type parameter* `a`, and allow the value in the box to be of that type `a`.

```
@dataclass
class Box[a]:
    value: a
```

This class is a generalized `Box` class that can be *specialized* into a specific `Box`. For example, by replacing `a` with `int` then we recover our `IntBox` class with an `int` value; replacing `a` with `str` recovers our `StrBox` class with a `str` value.

```
x: Box[int] = Box[int](1)
y: Box[str] = Box[str]('a')
z: Box[Box[int]] = Box(Box(1))
bad: Box[int] = Box[int]('a')
```

In Python and many Object-Oriented languages, `Box` is called a *generic* or *parametrically polymorphic* class/type. This is one example of a *type depending on a type*.

Polymorphic Functions

The same principle can be applied to *terms depending on types*. Suppose we have a function `singleton` that is to receive an object and puts that object in a list. In the same vein, we have to decide what the type of the parameter is, which dictates the corresponding return type. For example, may define this function that works on `int`s, and separately, another function that works on `str`s:

```
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]
```

Once again, we can observe that the implementations of these functions are identical, and only the types are different. Let us combine these implementations into a single function where the types are parameterized!

```
# Python 3.12
def singleton[a](x: a) -> list[a]:
    return [x]
x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

`singleton` is what is known as a *polymorphic function*: a function that depends on the type!

Polymorphic Functions in Haskell

How would we define the type of polymorphic functions in Haskell? That is pretty straightforward: type parameters are lowercase. For example, the `singleton` function can be defined like so:

```
singleton :: a -> [a]
singleton x = [x]
```

In fact we can see the type signatures of some built-in polymorphic functions:

```
ghci> :t head
head :: [a] -> a
ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Not sure what the type parameters are? Or, want to make your type parameters explicit? We can use `forall` to introduce a polymorphic function type, with the variables succeeding `forall` being the type parameters to the function.

```
ghci> :set -fprint-explicit-foralls
ghci> :t head
head :: forall a. [a] -> a
ghci> :t (.)
(.) :: forall b c a. (b -> c) -> (a -> b) -> a -> c
ghci> :{
ghci| singleton :: forall a. a -> [a]
ghci| singleton x = [x]
ghci| :}
ghci> singleton 2
[2]
ghci> singleton 'a'
"a"
```

Let's inspect the type signature of `(.)`. Recall that this function performs function composition; the implementation of `(.)` might look something like this:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) g f x = g (f x)
```

We have three terms, `g`, `f` and `x`. We know that `g` and `f` must be functions since we are calling them, thus we are going to let the types of `g` and `f` to be `d -> c` and `a -> b` respectively. Additionally, `x` is just some other term, and we will let its type be `e`. Thus for now, we shall let the type signature of `(.)` be the following, assuming the function ultimately returns `r`:

```
(.) :: (d -> c) -> (a -> b) -> e -> r
```

Now notice the following: for `f x` to be well-typed, the type of `x` must be the same as the type of the parameter to `f`, which is `a`. Thus, more accurately, `x` must be of type `a`:

```
(.) :: (d -> c) -> (a -> b) -> a -> r
```

We can now see that `f x` is well-typed, and this expression is of type `b`. We then pass this result into `g`. For this to be well-typed, again, the parameter type of `g` must match the type of `f x`. Thus, `g` must actually be of type `b -> c` for some `c`:

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

Finally, `g (f x)` has type `c`, which is what is returned from the function. As such, the return type of `(.) g f x` should also be `c`. This recovers the type signature shown by GHCI.

> You might be surprised to know that the process of recovering or reconstructing the types is known as type inference, which as stated in earlier chapters, is also done by

GHC! When you omit the type signature of any binding, GHC goes through this same process and helps us determine what the type of that binding is.

Programming with Polymorphic Types/Functions

When should we define polymorphic types or functions? As we have shown, when the implementations of classes, data types, functions etc. are the same except for the types, then we are able to parameterize the differing types which makes the class/data type/function polymorphic! Knowing immediately when to create polymorphic types/functions takes some practice, so to start, just create specialized versions of those types/functions, and as the need arises, make them polymorphic by parameterizing the appropriate types.

For example, suppose we are trying to create a `Tree` class that represents binary trees. Should this class be polymorphic? For now, let's ignore this fact and proceed to create a naive implementation of this class. Further suppose we are expecting to create a tree of integers, so we shall let that be the type of the values of our tree.

```
@dataclass
class IntTree:
    pass
@dataclass
class IntNode(IntTree):
    left: IntTree
    value: int
    right: IntTree
@dataclass
class IntLeaf(IntTree):
    value: int
```

Looks great! From this class we are able to create binary trees of integers, for example, `IntNode(IntLeaf(1), 2, IntLeaf(3))` gives a binary tree with preorder 2, 1 and 3.

Further suppose later on we need to store strings in a binary tree. Again, let's naively implement a separate class that does so:

```
@dataclass
class StrTree:
    pass
@dataclass
class StrNode(StrTree):
    left: StrTree
    value: str
    right: StrTree
@dataclass
class StrLeaf(StrTree):
    value: str
```

Once again, notice that the implementations of the classes are identical, and the only difference is in the types! This is one clear example where we should make our class polymorphic!

```
@dataclass
class Tree[a]:
    pass
@dataclass
class Node[a](Tree[a]):
    left: Tree[a]
    value: a
    right: Tree[a]
@dataclass
class Leaf[a](Tree[a]):
    value: a
```

Now from this one class, we are able to create all kinds of trees!

As another example, suppose we are trying to define a function that reverses a list. Once again, we have to be specific with the type of this function. Temporarily, we shall create a function that works on lists of integers:

```
def reverse_int(ls: list[int]) -> list[int]:
    return [] if not ls else \
           reverse_int(ls[1:]) + [ls[0]]
```

Then, later on we might have to define a similar function that reverses lists of strings:

```
def reverse_str(ls: list[str]) -> list[str]:
    return [] if not ls else \
           reverse_str(ls[1:]) + [ls[0]]
```

Once again, we can see that the implementations of the two functions are identical, and only the types are different. Make this function polymorphic!

```
def reverse[a](ls: list[a]) -> list[a]:
    return [] if not ls else \
           reverse(ls[1:]) + [ls[0]]
```

The two examples above give us some scenarios where we discover that we have to make a class or function polymorphic. More importantly, we see that the implementations across the specialized versions of the class/function are equal, and only the types differ. One key insight we can draw from this is: a class/function should be made polymorphic if its implementation is *independent* of the type(s) it is representing/acting on.

---

[1] These are the three axes that form the *lambda cube*, with the simply typed lambda calculus only having terms that depend on terms, and the Calculus of Constructions having types and terms depending on types and terms.

[2] The word *polymorphism* can be broken down into *poly* (many) and *morphism* (shape). The word is not just used in Computer Science, but in other areas like biology and pharmacology. Within Computer Science itself there are several kinds of polymorphism, and we shall investigate the most common ones in this lecture and in later lectures too. Finally, polymorphism in Computer Science is really about things taking on different forms, but I suspect that our description of parametric polymorphism gives a pretty good picture of what it entails.

LAST UPDATED **26 OCT 2024**

Algebraic Data Types

We have just seen different data types in Haskell, and introduced the concept of polymorphic types as demonstrated by examples in Python. Yet, we have not discussed how we can create our own (polymorphic) data types in Haskell!

Haskell is a purely functional language, so do not expect classes here. In OOP, objects have both data (attributes) and behaviour (methods), whereas this is not necessarily a principle in FP (although, you can have data types with functions as fields since functions are first-class). We already know how to create functions, so now we must investigate how we can create data types in a purely functional language.

If we think about it carefully, we might realize that data types are a mix of the following:

- A type **and** another type **and**...**and** yet another type
- A type **or** another type **or**...**or** yet another type

We can express the following types using **and** and **or** over other types:

- A `Fraction` consists of a numerator (`Int`) **and** a denominator (`Int`)
- A `Student` consists of a name (`String`) **and** an ID (`Int`)
- A `Bool` is either `True` **or** `False`
- A `String` is either an empty string **or** (a head character (`Char`) **and** a tail list (`String`))
- A polymorphic `Tree` is either (a leaf with a value of type `a`) **or** (a node with a value (`a`) **and** a left subtree (`Tree a`) **and** a right subtree (`Tree a`))

This formulation of data types as products (**and**) and/or sums (**sum**) is what is known as Algebraic Data Types (ADTs) (not to be confused with Abstract Data Types). In Haskell, types are **sums** of zero or more **constructors**; constructors are **products** of zero or more types.

To create a new data type in Haskell, we can use the `data` keyword. Let us create a fraction type based on our algebraic specification above:

```
data Fraction = Fraction Int Int

half :: Fraction
half = Fraction 1 2
```

On the left hand side we have the declaration of the type, and on the right hand side, a list of constructors separated by `|` that help us create the type. Note that the `Fraction` on the right hand side is the name of the constructor of the type; it in fact can be distinct from the name of the type itself (which is very helpful when you have more than one constructor). As you can see, to construct a `Fraction` (the type), the `Fraction` *constructor* receives two `Int`s, one numerator, and one denominator.

Then, defining the student type from our algebraic formulation above should also be straightforward:

```
data Student = S String Int

bob :: Student
bob = S "Bob" 123
```

Let us define the `Bool` type, which should have two constructors, each constructor not having any fields:

```
data Bool = True | False

true, false :: Bool
true = True
false = False
```

To construct a `Bool` we can use either the `True` constructor or the `False` constructor. Neither of these constructors receive any other fields.

We can also have multiple constructors that are products of more than zero types, as we shall see in the algebraic formulation of a `String`:

```
data String = EmptyString | Node Char String

hello, empty :: String
hello = Node 'h' (Node 'e' (Node 'l' (Node 'l' (Node 'o' EmptyString))))
empty = EmptyString
```

Polymorphic Algebraic Data Types

Now we show examples of creating our own polymorphic data types. The way we would do so is similar to how we defined generic/polymorphic classes in Python.

Let us start from the bottom again by creating specialized versions of a box type, this time in Haskell. We start by assuming that a box contains an `Int`:

```
data IntBox = IB Int
b :: IntBox
b = IB 1
```

Then define a box that contains a `String`:

```
data StrBox = SB String
b :: StrBox
b = SB "123"
```

Again, they look more or less the same, except for the type of the field. As such, we should allow `Box` to be polymorphic by introducing a type parameter:

```
data Box a = B a
x :: Box Int
x = B 1
y :: Box String
y = B "123"
```

Perfect! Let us try more complex polymorphic algebraic data types like linked lists and trees:

```
data LinkedList a = EmptyList | Node a (LinkedList a)
cat :: LinkedList Char
cat = Node 'c' (Node 'a' (Node 't' EmptyList))

data Tree a = Leaf a | TreeNode (Tree a) a (Tree a)
tree :: Tree Int
tree = TreeNode (Leaf 1) 2 (Leaf 3)
```

Constructors are actually functions!

```
ghci> data Fraction = F Int Int
ghci> :t F
F :: Int -> Int -> Fraction
ghci> :t F 1
F 1 :: Int -> Fraction
ghci> :t F 1 2
F 1 2 :: Fraction
```

We now have the facilities to define and construct data types and their terms, but so far we are not able to *access* the fields of a data type in Haskell. Unlike Python, we are not able to do something like `x.numerator` to obtain the numerator of a fraction `x`, for example. There are ways to define functions that do so and we will show them to you in later sections, but for now, Haskell has *record syntax* that automatically defines these accessor functions for us.

Let us re-create the `Student` type, this time using record syntax to automatically derive functions that obtain their names and IDs:

```
data Student = S { name :: String, id :: Int }
```

With this, we no longer need to define our own functions that access these fields for us. Record syntax is great for giving names to fields! Importantly, record syntax is nothing special, and we can continue to create terms of those types by way of usual constructor application.

```
x, y :: Student
x = S { name = "Alice", id = 123 }
y = S "Bob" 456
```

Let's try loading this into GHCI and see the accessor functions in action:

```
ghci> name x
"Alice"
ghci> id y
456
```

You can also make use of record syntax to express record updates. For example, we can update Alice to have the ID of 456 like so:

```
ghci> id x
123
ghci> z = x { id = 456 }
ghci> name z
"Alice"
ghci> id z
456
```

Of course, the original term was not actually *updated* since everything is immutable in Haskell— `x { id = 456 }` simply constructs a new term that contains the same values for all its fields, except where the `id` field now takes the value `456`.

We can even mix and match these different forms of constructor definitions, or create large data structures!

```
data Department = D {name' :: String, courses :: [Course]}
data Course = C { code :: String,
                  credits :: Int,
                  students :: [Student] }
data Student = UG { homeFac :: String,
                    name :: String,
                    id :: Int }
             | PG [String] String Int

alice   = UG "SoC" "Alice" 123
bob     = PG ["SoC", "YLLSoM"] "Bob" 456
it5100a = C "IT5100A" 2 [alice]
it5100b = C "IT5100B" 2 [alice, bob]
cs      = D "Computer Science" [it5100a, it5100b]
```

More on Polymorphism

Now that we have shown how to create our own algebraic data types in Haskell (and polymorphic ones), we step aside and give a mental model for understanding polymorphism. Recall that we have described polymorphic functions and types as functions/types that quantifies/parameterizes types; in other words, they receive a type as a parameter.

Recall in the lambda calculus that λ creates a function over a parameter. Assuming the parameter has type S and the returned value has type T, we get:

$$\lambda \mathrm{x}.\, \mathrm{e} : \mathrm{S} \to \mathrm{T}$$

and when we call or apply this function, we are substituting the parameter for the argument of the function application:

$$(\lambda \mathrm{x}.\, \mathrm{e}_1)\mathrm{e}_2 \equiv_\beta \mathrm{e}_1[\mathrm{x} := \mathrm{e}_2]$$

$$\begin{aligned}(\lambda \mathrm{x} : \mathtt{Int}.\, \mathrm{x} + 4)3 &\equiv_\beta (\mathrm{x} + 4)[\mathrm{x} := 3] \\ &\equiv_\beta (3 + 4) \\ &\equiv_\beta 7\end{aligned}$$

In Haskell (the expression in parentheses is a lambda expression):

```
ghci> (\x -> x + 4) 3
7
```

A typed variant of the lambda calculus known as System F has polymorphic functions, which are functions that also receive a type parameter. We can then apply this function onto a type *argument* to get a specialized version of that function. Such type parameters are bound by Λ. As an example, if we have a term e of type T, we get:

$$\Lambda\alpha.\, \mathrm{e} : \forall\alpha.\, \mathrm{T}$$

Calling or applying this function with a type argument, once again, substitutes the type parameter with the type argument:

$$(\Lambda\alpha.\, \mathrm{e})\ \tau \equiv_\beta \mathrm{e}[\alpha := \tau]$$

$$(\Lambda\alpha.\, \mathrm{e})\ \tau : \mathrm{T}[\alpha := \tau]$$

$$\begin{aligned}(\Lambda\alpha.\, \lambda \mathrm{x} : \alpha.\, [\mathrm{x}])\mathtt{Int} &\equiv_\beta (\lambda \mathrm{x} : \alpha.\, [\mathrm{x}])[\alpha := \mathtt{Int}] \\ &\equiv_\beta \lambda \mathrm{x} : \mathtt{Int}.\, [\mathrm{x}]\end{aligned}$$

We can show this with an example in Haskell. Explicit type arguments must be enabled with a language extension and the type arguments must be prefixed by `@`:

```
ghci> :set -XTypeApplications -fprint-explicit-foralls
ghci> :{
ghci| f :: forall a. a -> [a]
ghci| f x = [x]
ghci| :}

ghci> :t f
f :: forall a. a -> [a]

ghci> :t f @Int
f @Int :: Int -> [Int]

ghci> f @Int 1
[1]
```

On the other hand, polymorphic types can be seen as *functions at the type-level*. These are "functions" that receive types and return types! For example, we can define a `Pair` type that is polymorphic in its component types. Thus, the `Pair` type itself (not its constructor!) receives two types, and returns the resulting `Pair` type specialized to those component types. This makes `Pair` what is known as a *type constructor*.

To observe this fact, know that types are to terms as *kinds* are to types: they describe what *kind* of type a type is. The usual types that we encounter `Int`, `[[Char]]` etc. have kind `*`, and type constructors or "type-level functions" have kind `* -> *` for example. Below, we show that `Pair` is a type constructor of kind `* -> * -> *`, which makes sense since it receives two types and returns the specialized type of the `Pair`:

```
ghci> data Pair a b = P a b
ghci> :k Pair
Pair :: * -> * -> *
ghci> :k Pair Int
Pair Int :: * -> *
ghci> :k Pair Int String
Pair Int String :: *
```

We know that we can have higher-order functions, for example, the type of `map` might be something like `(a -> b) -> [a] -> [b]`. Can we have higher-order type constructors? Yes! These are known as *higher kinds* or *higher-kinded types*. These types receive *type constructors* as type arguments. Let us construct a higher-kinded type that receives a type constructor and applies it onto a type:

```
ghci> data Crazy f a = C (f a)
```

Upon visual inspection we can see that `f` must be a type constructor, because the constructor `C` receives a term of type `f a`! What's crazier is, when inspecting the kind of `Crazy`, we see that it exhibits *kind polymorphism*:

```
ghci> :set -fprint-explicit-foralls
ghci> :k Crazy
Crazy :: forall {k}. (k -> *) -> k -> *
```

To give you an example of how this might work, because we know we can construct lists of any type, `[]` (the type, not the empty list) must be a type constructor. We can thus pass the `[]` type constructor into `Crazy`:

```
ghci> :k Crazy []
Crazy [] :: * -> *
ghci> :k Crazy [] Int
Crazy [] Int :: *
```

How might this work? We see that `Crazy [] Int` has kind `*`, so we should be able to construct a term of this type. We can do so by using the `C` constructor defined above! To be clear, let's see the specialized version of the constructor with the type arguments entered:

```
ghci> :t C @[] @Int
C @[] @Int :: [Int] -> Crazy [] Int
```

As we can see, to construct a term of this type, we just need to pass in a list of integers to `C`:

```
ghci> x :: Crazy [] Int = C [1]
```

We can in fact instantiate other crazy types with different type constructors:

```
ghci> data Box a = B a
ghci> y :: Crazy Box Int = C (B 2)
```

The utility of higher-kinded types may not be apparent to you now; later on we might see some of them in action!

Although this might confuse you so far, what we have demonstrated merely serves to demonstrate the idea that parametric polymorphism can be thought of the phenomenon where something (type or term) can receive a type and give you a type or term, just as we have stated at the beginning of Chapter 2.2 (Polymorphism).

Other Polymorphisms

At the start of Chapter 2.2 (Polymorphism) we introduced three questions, two of which have been answered. Let us restate the final question and pose one more:

1. Can types depend on terms?
2. Are there other kinds of polymorphism?

The answers to both questions is yes. Types that depend on terms are known as *dependent types*, which we shall not cover in this course. There are also other kinds of polymorphisms, some of which you have already dealt with. Subtype polymorphism is used frequently in OOP, since subclasses are types that are *subtypes* of their superclasses. An umbrella term *ad-hoc polymorphism* generally refers to *overloading*, which we shall discuss in the future. There are also more kinds of polymorphisms, but we shall not discuss them in this course.

Python (and several other mainstream languages) is quite special, being a multi-paradigm language means that several forms of polymorphism are applicable to it. In particular, we have seen that Python supports parametric polymorphism, and since Python supports OOP, it also has subtype polymorphism. Despite Python not having algebraic data types (yet), we may also formulate our types to behave similarly to Algebraic Data Types. Two formulations we may attempt are: 1) with types as unions and constructors as classes, 2) with types as classes and constructors as their subclasses. Below we present both formulations for the linked list type:

```
# (1)
type List[a] = Node[a] | Empty

@dataclass
class Empty:
    pass

@dataclass
class Node[a]:
    head: a
    tail: List[a]

x: List[int] = Node(1, Node(2, Empty()))


# (2)
from typing import Any
@dataclass
class List[a]:
    pass

@dataclass
class Empty(List[Any]):
    pass

@dataclass
class Node[a](List[a]):
    head: a
    tail: List[a]

x: List[int] = Node(1, Node(2, Empty()))
```

There are some differences between the two formulations, and between these with Haskell's Algebraic Data Types. Most importantly, in Haskell, data types are types, but constructors are not. This is unlike Python, where all classes are types. That means a

variable of type `Node[int]` is valid in Python, but a variable of type `Node Int` is not in Haskell.

Generalized Algebraic Data Types

However, something interesting is going on here. In the second formulation, a `Node[a]` is a `List[a]`, which makes sense. On the other hand, an `Empty` can be typed as `List[Any]`, because an empty list fits all kinds of lists. An interesting observation you might see is that the supertype of our "constructors" need not strictly be `List[a]`, it could be any kind of list!

Consider the following example of defining simple expressions in a programming language, which is defined polymorphically using OOP:

```
class Expr[a]:
    def eval(self) -> a:
        raise Exception
```

The `Expr` class is parameterized by the type of its evaluation. From this class we may now create subclasses of `Expr`. For example, some simple numeric expressions.

```
@dataclass
class LitNumExpr(Expr[int]):
    n: int
    def eval(self) -> int:
        return self.n

@dataclass
class AddExpr(Expr[int]):
    lhs: Expr[int]
    rhs: Expr[int]
    def eval(self) -> int:
        return self.lhs.eval() + self.rhs.eval()
```

We can then create other kinds of expressions. For example, an equality expression that returns booleans:

```
@dataclass
class EqExpr[a](Expr[bool]):
    lhs: Expr[a]
    rhs: Expr[a]
    def eval(self) -> bool:
        return self.lhs.eval() == self.rhs.eval()
```

Or even a conditional expression whose evaluated type is parameterized:

```
@dataclass
class CondExpr[a](Expr[a]):
    cond: Expr[bool]
    true: Expr[a]
    false: Expr[a]
    def eval(self) -> a:
        return self.true.eval() if self.cond.eval() else self.false.eval()
```

Let's try this out! Suppose we would like to evaluate the following expression:

```
if 1 == 2 then 1 + 1 else 0
```

Let's write this in the program using our classes and evaluate it!

```
zero: Expr[int] = LitNumExpr(0)
one: Expr[int] = LitNumExpr(1)
two: Expr[int] = LitNumExpr(2)
one_plus_one: Expr[int] = AddExpr(one, one)
one_eq_two: Expr[bool] = EqExpr(one, two)
cond: Expr[int] = CondExpr(one_eq_two, one_plus_one, zero)
print(cond.eval()) # 0
```

How do we create such an algebraic data type in Haskell? For this, we have to use *Generalized Algebraic Data Types* (GADTs). Loosely, these are algebraic data types like before, except that each constructor can decide what type it returns!

First, let us formulate our original algebraic data types using GADT syntax.

```
data LinkedList a where
    EmptyList :: LinkedList a -- this is a different a!
    Node :: b -> LinkedList b -> LinkedList b
```

Now let us take it a step further, and truly customize the constructors of an `Expr` GADT:

```
data Expr a where
    LitNumExpr :: Int -> Expr Int
    AddExpr    :: Expr Int -> Expr Int -> Expr Int
    EqExpr     :: Expr a -> Expr a -> Expr Bool
    CondExpr   :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Pretty neat huh! There are many uses of GADTs, and we might see them in the future. In the next section, we will show you how we can write functions against algebraic data types and GADTs, including how we can implement the `eval` function.

LAST UPDATED 26 OCT 2024

Pattern Matching

We have seen how we can write constructors for algebraic data types, and even use record syntax to create functions for accessing fields. However, one natural question would then be to ask, how do we write functions that access these fields, if we do not use record syntax? For example, if we defined a fraction type normally, how do we obtain a fraction's numerator and denominator?

The answer to this question is to use *pattern matching*. It is a control structure just like `if` - `then` - `else` expressions, except that we would execute different branches based on the *value/structure* of the data, instead of a general condition.

Let us define the `factorial` function using pattern matching instead of conditional expressions or guards. We use `case` expressions to do so:

```
fac :: Int -> Int
fac n = case n of -- match n against these patterns:
    0 -> 1
    x -> x * fac (x - 1) -- any other Int
```

The nice thing about pattern matching is that we can also match against the *structure* of data, i.e. to match against constructors. Let us redefine the `fst` and `snd` functions which project a pair into its components:

```
fst' :: (a, b) -> a
fst' p = case p of
    (x, _) -> x

snd' :: (a, b) -> b
snd' p = case p of
    (_, y) -> y
```

Let us also write accessor functions to access the numerator and denominator of a fraction.

```
data Fraction = F Int Int
numerator, denominator :: Fraction -> Int
numerator f = case f of
    F x _ -> x
denominator f = case f of
    F _ x -> x
```

One nice thing about Haskell is that because we perform pattern matching over the arguments of functions so frequently, we can actually bring the patterns up to the

definitions of the functions themselves. Let us define all the functions we've just written using `case` expressions into more idiomatic uses of pattern matching.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)

fst' :: (a, b) -> a
fst' (x, _) = x

snd' :: (a, b) -> b
snd' (_, y) = y

data Fraction = F Int Int
numerator, denominator :: Fraction -> Int

numerator (F x _) = x
denominator (F _ y) = y
```

We also know that the list type is a singly linked list, which is roughly defined as such:

```
data [a] = [] | a : [a]
```

We can use this fact to pattern match against lists! For instance, the sum of a list of integers is 0 if the list is empty, otherwise its the head of the list plus the sum of the tail of the list.

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

Similarly, the length of a list is 0 if the list is empty, otherwise it is 1 more than the length of its tail.

```
len :: [a] -> Int
len [] = 0
len (_ : xs) = 1 + len xs
```

Really neat! Defining functions operating on algebraic data types (including recursive data types) are very convenient thanks to pattern matching! What's more, patterns can actually be used virtually anywhere on the left side of any binding:

Let us use pattern matching in a `let` binding:

```
len :: [a] -> Int
len [] = 0
len ls =
    let (_ : xs) = ls
    in  1 + len xs
```

Perhaps the most powerful feature of pattern matching is that the compiler will warn you if your pattern matches are non-exhaustive, i.e. if you do not match against all possible constructors of the type! Let us define a function that only matches against the empty list constructor.

```
-- Main.hs
emp :: [a] -> [a]
emp [] = []
```

Compile it to see the warning!

```
ghc Main.hs
```

```
Main.hs:3:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'emp': Patterns of type '[a]' not matched: (_:_)
  |
3 | emp [] = []
  | ^^^^^^^^^^
```

This is one reason why pattern matching is so powerful: compilers can check if you have covered all possible patterns of a given type. This is unlike the usual `if` - `else` statements in other languages where it is much less straightforward to check if you have covered all possible branches, especially if you omit `else` statements.

One important point to highlight here is that pattern matching is done top-down. Pattern-matching is kind of similar to `if` - `else` statements in that regard: your most specific condition should be defined first, then followed by more general or catch-all patterns.

The following factorial function is poorly defined, because the first pattern match will match all possible integers, thereby causing the function to never terminate:

```
fac :: Int -> Int
fac n = n * fac (n - 1)
fac 0 = 1 -- redundant as pattern above matches all possible integers
```

With pattern matching, let us know fulfil our earlier promise of defining the `eval` function for the `Expr` GADT in Chapter 2.3 (Algebraic Data Types). In our Python formulation, we know that `eval` should have the type signature `Expr a -> a`. Let us then define how each expression should be evaluated with pattern matching.

```
-- Main.hs
eval :: Expr a -> a
eval (LitNumExpr n)   = n
eval (AddExpr a b)    = eval a + eval b
eval (EqExpr a b)     = eval a == eval b
eval (CondExpr a b c) = if eval a then eval b else eval c
```

This seems straightforward. However, you might find that when this program is compiled, the compiler throws an error on the use of the `(==)` function:

```
ghc Main.hs
```

```
Main.hs:13:28: error:
    • Could not deduce (Eq a1) arising from a use of ‘==’
      from the context: a ~ Bool
        bound by a pattern with constructor:
                   EqExpr :: forall a. Expr a -> Expr a -> Expr Bool,
                 in an equation for ‘eval’
        at app/Main.hs:13:7-16
      Possible fix:
        add (Eq a1) to the context of the data constructor ‘EqExpr’
    • In the expression: eval a == eval b
      In an equation for ‘eval’: eval (EqExpr a b) = eval a == eval b
   |
13 | eval (EqExpr a b) = eval a == eval b
   |
```

The reason for this is Haskell is unable to determine that the type parameter `a` is amenable to equality comparisons. Solving this requires an understanding of *typeclasses*, which we will explore in the next chapter. For now, just include an `Eq a =>` constraint in our GADT declaration.

You might also get a warning about pattern matching on GADTs being fragile; that is because GADTs are actually a Haskell language extension. As such, enable this extension when compiling this program, or add a `LANGUAGE` *pragma* at the top of the file.

```
{-# LANGUAGE GADTs #-}
data Expr a where
  LitNumExpr ::         Int -> Expr Int
  AddExpr    ::         Expr Int -> Expr Int -> Expr Int
  EqExpr     :: Eq a => Expr a -> Expr a -> Expr Bool
  CondExpr   ::         Expr Bool -> Expr a -> Expr a -> Expr a
```

Our program should compile now!

Pattern Matching in Python

Python also has pattern matching with `match` statements with `case` clauses! It looks very similar to how we would write `case` expressions in Haskell.

```
def factorial(n: int) -> int:
  match n:
    case 0: return 1
    case n: return n * factorial(n - 1)
```

We can also match on the structure of types by unpacking. For example, defining a function that sums over a list of integers:

```
def sum(ls: list[int]) -> int:
  match ls:
    case []: return 0
    case (x, *xs): return x + sum(xs)
    case _: raise TypeError()
```

Alternatively, performing structural pattern matching over a so called algebraic data type:

```
@dataclass
class Tree[a]: pass

@dataclass
class Node[a](Tree[a]):
    val: a
    left: Tree[a]
    right: Tree[a]

@dataclass
class Leaf[a](Tree[a]):
    val: a

def preorder[a](tree: Tree[a]) -> list[a]:
    match tree:
        case Node(v, l, r): return [v] + preorder(l) + preorder(r)
        case Leaf(v): return [v]
        case _: raise TypeError
```

However, notice that in the `sum` and `preorder` function definitions, the last clause catches all patterns and raises an error. This is needed to side-step the exhaustiveness checker. This is because we are using classes to model algebraic data types, and Python does not always know all the possible structures of a given class. In the case of `sum`, Python's type system does not contain information about the length of a list, so it has no way of determining exhaustiveness. In the case of `preorder`, the reason omitting the last case gives a non-exhaustiveness error is because we did not match against other possible subclasses of `Tree`.

If we had formulated our `Tree` type using unions, `pyright` can determine the exhaustiveness of our patterns:

```
type Tree[a] = Node[a] | Leaf[a]

@dataclass
class Node[a]:
    val: a
    left: Tree[a]
    right: Tree[a]

@dataclass
class Leaf[a]:
    val: a

def preorder[a](tree: Tree[a]) -> list[a]:
    match tree:
        case Node(v, l, r): return [v] + preorder(l) + preorder(r)
        case Leaf(v): return [v]
        # no need for further cases
```

However, this may not always be ideal, especially if we are to define GADTs in Python. Until Algebraic Data Types or ways to annotate the exhaustivity of subclasses (such as defining a *sealed* class) are formally introduced, exhaustive pattern matching checks are going to be difficult to do. When doing pattern matching in Python, ensure that all possible cases are handled before doing a catch-all clause in your `match` statement.

All-in-all, we have just introduced a new control structure known as pattern matching. When should we use this control structure? The general rule of thumb is as follows:

- If you are doing different things based on the value and/or structure of data, use pattern matching. You can tell this is the case if you are doing equality and `isinstance` checks in your conditional statements in Python.
- Otherwise, you are likely going with the more general case of doing different things based on the satisfiability of a condition, in which case, rely on `if-else` statements, or in Haskell, conditional expressions and/or guards.

LAST UPDATED **26 OCT 2024**

Exercises

Question 1

Without using GHCI, determine the types of the following expressions:

1. `(1 :: Int) + 2 * 3`
2. `let x = 2 + 3 in show x`
3. `if "ab" == "abc" then "a" else []`
4. `(++ [])`
5. `map (\(x :: Int) -> x * 2)`
6. `((\(x :: [Int]) -> show x) . )`
7. `( . (\(x :: [Int]) -> show x))`
8. `(,) . fst`
9. `filter`

Question 2

Without the help of GHCI, describe the types of `eqLast`, `isPalindrome`, `burgerPrice` and `(@:)` which we defined in Chapter 1.4 (Course Introduction#Exercises)

Question 3

Recall the following definition of `burgerPrice`:

```
burgerPrice burger
  | null burger = 0
  | otherwise   =
      let first = ingredientPrice (head burger)
          rest  = burgerPrice (tail burger)
      in  first + rest
  where ingredientPrice i
          | i == 'B' = 0.5
          | i == 'C' = 0.8
          | i == 'P' = 1.5
          | i == 'V' = 0.7
          | i == 'O' = 0.4
          | i == 'M' = 0.9
```

There are several problems with this. First of all, writing `burgerPrice` with guards does not allow us to rely on compiler exhaustiveness checks, and may give us some additional warnings about `head` and `tail` being *partial*, despite their use being perfectly fine. The second problem is that we have allowed our burger to be any string, even though we should only allow strings that are composed of valid ingredients—the compiler will not reject invocations of `burgerPrice` with bogus arguments like `"AbcDEF"`.

Define a new type that represents valid burgers, and re-define `burgerPrice` against that type using pattern matching. Additionally, provide a type declaration for this function. Note that you may use the `Rational` type to describe rational numbers like `0.8` etc, instead of `Double` which may have precision issues. You might see that the output of your `burgerPrice` function is of the form `x % y` which means x/y.

Question 4

Define a function `dropConsecutiveDuplicates` that receives a list of any type that is amenable to equality comparisons and removes all the consecutive duplicates of the list. Example runs follow:

```
ghci> dropConsecutiveDuplicates []
[]
ghci> dropConsecutiveDuplicates [1, 2, 2, 3, 3, 3, 3, 4, 4]
[1, 2, 3, 4]
ghci> dropConsecutiveDuplicates "aabcccddeee"
"abcde"
```

For this function to be polymorphic, you will need to add a constraint `Eq a =>` at the beginning of the function's type signature just like we did for the `EqExpr` constructor of our `Expr a` GADT.

Question 5

Suppose we have a list `[1,2,3,4,5]`. Since lists in Haskell are singly-linked lists, and not to mention that Haskell lists are immutable, changing the values at the tail end of the list (e.g. `4` or `5`) can be inefficient! Not only that, if we want to then change something near the element we've just changed, we have to traverse all the way down to that element from the head all over again!

Instead, what we can use is a *zipper*, which allows us to focus on a part of a data structure so that accessing those elements and walking around it is efficient. The idea is to write functions that let us walk down the list, do our changes, and walk back up to recover the full list. For this, we shall define some functions:

1. `mkZipper` which receives a list and makes a zipper
2. `r` which walks to the right of the list zipper
3. `l` which walks to the left of the list zipper
4. `setElement x` which changes the element at the current position of the zipper to `x`.

Example runs follow:

```
ghci> x = mkZipper [1,2,3,4,5]
ghci> x
([], [1,2,3,4,5])
ghci> y = r $ r $ r $ r x
ghci> y = ([4,3,2,1], [5])
ghci> z = setElement (-1) y
ghci> z
([4,3,2,1], [-1])
ghci> w = setElement (-2) $ l z
ghci> w
([3,2,1], [-2,-1])
ghci> l $ l $ l w
([], [1,2,3,-2,-1])
```

Question 6

Let us create a data structure that represents sorted sets. These are collections that contain unique elements and are sorted in ascending order. A natural data structure that can represent such sets is the Binary Search Tree (BST) abstract data type (ADT).

Create a new type `SortedSet`. Then define the following functions:

1. The function `@+` that receives a sorted set and an element, and returns the sorted set with the element added (unless it is already in the sorted set).
2. The function `setToList` that receives a sorted set and returns it as a list (in sorted order)

3. The function `sortedSet` that receives a list of elements and puts them all in a sorted set.
4. The function `in'` which determines if an element is in the sorted set.

Note that if any of your functions perform any comparison operations (`>` etc.), you will need to include the `Ord a =>` constraint over the elements of the sorted set or list at the beginning of the type signature of those functions. Example runs follow:

```
ghci> setToList $ (sortedSet []) @+ 1
[1]
ghci> setToList $ (sortedSet []) @+ 1 @+ 2
[1,2]
ghci> setToList $ (sortedSet []) @+ 1 @+ 2 @+ 0
[0,1,2]
ghci> setToList $ (sortedSet []) @+ 1 @+ 2 @+ 0 @+ 2
[0,1,2]
ghci> setToList $ sortedSet [7,3,2,5,5,2,1,7,6,3,4,2,4,4,7,1,2,3]
[1,2,3,4,5,6,7]
ghci> setToList $ sortedSet "aaabccccbbbbbaaaaab"
"abc"
ghci> 1 `in'` (sortedSet [1, 2, 3])
True
ghci> 1 `in'` (sortedSet [4])
False
```

Question 7

In this question, we are going to demonstrate an example of the *expression problem* by writing FP-style data structures and functions, and OO-style classes, to represent the same problem. We shall use Haskell for the FP formulation, and Python for the OOP formulation. Ensure that your Python code is well-typed by checking it with `pyright` .

The problem is as such. We want to represent various shapes, and the facility to calculate the area of a shape. To start, we shall define two shapes: circles and rectangles. Circles have a radius and rectangles have a width and height. Assume these fields are all `Double` s in Haskell, and `float` s in Python.

- Haskell: define a type `Shape` that represents these two shapes, and a function `area` that computes the area of any shape.
- Python: define a (abstract) class `Shape` that comes with a (abstract) method `area` which gives its area. Then, define two subclasses of `Shape` that represents circles and rectangles, and define their constructors and methods appropriately.

The *expression problem* essentially describes the phenomenon that it can either be easy to add new representations of a type or easy to add new functions over types, but not both. To observe this, we are going to extend the code we've written in the following ways:

1. Create a new shape called `Triangle` that has a width and height.
2. Create a new function/method `scale` that scales the shape (by length) by some factor `n`.

Proceed to do so in both formulations. As you are doing so, think about whether each extension is easy to do if the code we've previously written cannot be amended, e.g. if it is in a pre-compiled library which you do not have the source code of.

Question 8

Let us extend our Expressions GADT. Define the following expressions:

1. `LitBoolExpr` holds a boolean value (`True` or `False`)
2. `AndExpr` has two boolean expressions and evaluates to their conjunction
3. `OrExpr` has two boolean expressions and evaluates to their disjunction
4. `FuncExpr` holds a function
5. `FuncCall` receives a function and an argument, and evaluates to the function application to that argument

Example runs follow:

```
ghci> n = LitNumExpr
ghci> b = LitBoolExpr
ghci> a = AndExpr
ghci> o = OrExpr
ghci> f = FuncExpr
ghci> c = FuncCall
ghci> eval (b True `a` b False)
False
ghci> eval (b True `a` b True)
True
ghci> eval (b True `o` b False)
True
ghci> eval (b False `o` b False)
False
ghci> eval $ f (\x -> x + 1) `c` n 1
2
ghci> eval $ c (c (f (\x y -> x + y)) (n 1)) (n 2)
3
```

Question 9

In this question we shall simulate a simple banking system consisting of bank accounts. We shall write all this code in **Python**, but in a typed functional programming style. That means:

1. No loops
2. No mutable data structures or variables
3. Pure functions only
4. Annotate all variables, functions etc. with types
5. Program must be type-safe

There are several kinds of bank accounts that behave differently on certain operations. We aim to build a banking system that receives such operations that act on these accounts. We shall build this system incrementally (as we should!), so you may want to follow the parts in order, and check your solutions after completing each part.

Bank Accounts

Bank Account ADT

First, create an Algebraic Data Type (ADT) called `BankAccount` that represents two kinds of bank accounts:

1. Normal bank accounts
2. Minimal bank accounts

Both kinds of accounts have an ID, account balance and an interest rate.

Example runs follow:

```
>>> NormalAccount("a", 1000, 0.01)
NormalAccount(account_id='a', balance=1000, interest_rate=0.01)
>>> MinimalAccount("a", 1000, 0.01)
MinimalAccount(account_id='a', balance=1000, interest_rate=0.01)
```

Basic Features

Now let us write some simple features of these bank accounts. There are two features we shall explore:

1. Depositing money into a bank account. Since we are writing code in a purely functional style, our function does not mutate the state of the bank account. Instead, it returns a new state of the account with the money deposited. Assume that the deposit amount is non-negative.
2. Deducting money from a bank account. Just like before, we are not mutating the state of the bank account, and instead will be returning the new state of the bank account. However, the deduction might not happen since the account might have insufficient funds. As such, this function returns a tuple containing a boolean flag describing whether the deduction succeeded, and the new state of the bank account after the

deduction (if the deduction does not occur, the state of the bank account remains unchanged).

Note: The type of a tuple with two elements of types `A` and `B` is `tuple[A, B]`. Example runs follow:

```
>>> x = NormalAccount('abc', 1000, 0.01)
>>> y = MinimalAccount('bcd', 2000, 0.02)
>>> deposit(1000, x)
NormalAccount(account_id='abc', balance=2000, interest_rate=0.01)
>>> deduct(1000, x)
(True, NormalAccount(account_id='abc', balance=0, interest_rate=0.01))
>>> deduct(2001, y)
(False, MinimalAccount(account_id='bcd', balance=2000,
    interest_rate=0.02))
```

Advanced Features

Now we shall implement some more advanced features:

1. Compounding interest. Given a bank account with balance b and interest rate i, the new balance after compounding will be $\mathrm{b}(1 + \mathrm{i})$. For minimal accounts, an administrative fee of $20 will be deducted if its balance is strictly below $1000. This fee deduction happens **before** compounding. Importantly, bank balances never go below $0, so e.g. if a minimal account has $10, after compounding, its balance will be $0.

2. Bank transfers. This function receives a transaction amount and two bank accounts: (1) the credit account (the bank account where funds will come from) and (2) the debit account (bank account where funds will be transferred to). The result of the transfer is a triplet (tuple of three elements) containing a boolean describing the success of the transaction, and the new states of the credit and debit accounts. The transaction does not happen if the credit account has insufficient funds.

Example runs follow:

```
>>> x = NormalAccount('abc', 1000, 0.01)
>>> y = MinimalAccount('bcd', 2000, 0.02)
>>> z = MinimalAccount('def', 999, 0.01)
>>> w = MinimalAccount('xyz', 19, 0.01)
>>> compound(x)
NormalAccount(account_id='abc', balance=1010, interest_rate=0.01)
>>> compound(compound(x))
NormalAccount(account_id='abc', balance=1020.1, interest_rate=0.01)
>>> compound(y)
MinimalAccount(account_id='bcd', balance=2040, interest_rate=0.02)
>>> compound(z)
MinimalAccount(account_id='def', balance=988.79, interest_rate=0.01)
>>> compound(w)
MinimalAccount(account_id='xyz', balance=0, interest_rate=0.01)
>>> transfer(2000, x, y)
(False, NormalAccount(account_id='abc', balance=1000,
    interest_rate=0.01), MinimalAccount(account_id='bcd',
    balance=2000, interest_rate=0.02))
>>> transfer(2000, y, x)
(True, MinimalAccount(account_id='bcd', balance=0,
    interest_rate=0.02), NormalAccount(account_id='abc',
    balance=3000, interest_rate=0.01))
```

Operating on Bank Accounts

Let us suppose that we have a dictionary whose keys are bank account IDs and values are their corresponding bank accounts. This dictionary simulates a 'database' of bank accounts which we can easily lookup by bank account ID:

```
>>> d: dict[str, BankAccount] = {
  'abc': NormalAccount('abc', 1000, 0.01)
  'bcd': MinimalAccount('bcd', 2000, 0.02)
}
```

Now we are going to process a whole bunch of operations on this 'database'.

Operations ADT

The first step in processing a bunch of operations on the accounts in our database is to create a data structure that represents the desired operation in the first place. For this, create an algebraic data type `Op` comprised of two classes:

1. `Transfer` : has a transfer amount, and credit bank account ID, and a debit bank account ID. This represents the operation where we are transferring the transfer amount from the credit account to the debit account.
2. `Compound` . This just tells the processor to compound all the bank accounts in the map. There should be no attributes in this class.

Processing One Operation

Write a function `process_one` that receives an operation and a dictionary of bank accounts (keys are bank account IDs, and values are the corresponding bank accounts), and performs the operation on the bank accounts in the dictionary. As a result, the function returns a pair containing:

1. A boolean value to describe whether the operation has succeeded
2. The resulting dictionary containing the updated bank accounts after the operation has been processed.

Take note that there are several ways in which a `Transfer` operation may fail:

1. If any of the account IDs do not exist in the dictionary, the transfer will fail
2. If the credit account does not have sufficient funds, the transfer will fail
3. Otherwise, the transfer should proceed as per normal

Keep in mind that you should not mutate any data structure used. Example runs follow:

```
# data
>>> alice = NormalAccount('alice', 1000, 0.1)
>>> bob = MinimalAccount('bob', 999, 0.1)
>>> mp = {'alice': alice, 'bob': bob}

# ops
>>> c = Compound()
>>> t1 = Transfer(1000, 'alice', 'bob')
>>> t2 = Transfer(1000, 'bob', 'alice')

# processing compound operation
>>> process_one(c, mp)
(True, {'alice': NormalAccount('alice', 1100.0, 0.1),
        'bob': MinimalAccount('bob', 1076.9, 0.1)})

# processing transfers
>>> process_one(t1, mp)
(True, {'alice': NormalAccount('alice', 0, 0.1),
        'bob': MinimalAccount('bob', 1999, 0.1)})
>>> process_one(t2, mp)
(False, {'alice': NormalAccount('alice', 1000, 0.1),
         'bob': MinimalAccount('bob', 999, 0.1)})
```

Processing All Operations

Now let us finally define a function `process_all` that receives a list of operations and a dictionary of bank accounts (the keys are bank account IDs, and the values are bank accounts). As a result, the function returns a pair containing:

1. A list of booleans where the i^{th} boolean value describes whether the i^{th} operation has succeeded

2. The resulting dictionary containing the updated bank accounts after all the operations have been processed.

Example runs follow:

```
# data
>>> alice = NormalAccount('alice', 1000, 0.1)
>>> bob = MinimalAccount('bob', 999, 0.1)
>>> mp = {'alice': alice, 'bob': bob}

# op
>>> c = Compound()
>>> t1 = Transfer(1000, 'alice', 'bob')
>>> t2 = Transfer(1000, 'bob', 'alice')

# process
>>> process_all([t2, c, t2, t1], mp)
([False, True, True, True],
 {'alice': NormalAccount(account_id='alice', balance=1100.0,
interest_rate=0.1),
  'bob': MinimalAccount(account_id='bob', balance=1076.9, interest_rate=0.1)})
```

Polymorphic Processing

Let us assume that your `process_all` function invokes the `process_one` function. If you were careful with your implementation of `process_all`, you *should* be able to lift your `proces_one` function as a parameter:

```
def process_all(ops, mp):
    # ...
    process_one(...)
    # ...

# becomes

def process_all(f, ops, mp):
    # ...
    f(...)
    # ...
```

After which, nothing about the implementation of `process_all` depends on the types like `Op`, `dict[str, BankAccount]` or `bool`. Thus, we should make this function polymorphic!

Our goal is to write a polymorphic function `process` that can process any list over a state and produce the resulting list and an updated state after performing stateful processing over the list. It should be defined such that `process(process_one, ops, mp)` should be the exact same as `process_all(ops, mp)` as you have defined earlier:

```
# data
>>> alice = NormalAccount('alice', 1000, 0.1)
>>> bob = MinimalAccount('bob', 999, 0.1)
>>> mp = {'alice': alice, 'bob': bob}

# ops
>>> c = Compound()
>>> t1 = Transfer(1000, 'alice', 'bob')
>>> t2 = Transfer(1000, 'bob', 'alice')

# process
>>> process(process_one, [t2, c, t2, t1], mp)
([False, True, True, True],
 {'alice': NormalAccount(account_id='alice', balance=1100.0,
interest_rate=0.1),
  'bob': MinimalAccount(account_id='bob', balance=1076.9, interest_rate=0.1)})
```

Furthermore, the best part of this polymorphic function is that it can be used in any situation where we need this stateful accumulation over a list. For example, we can define a function that tests if a number n is co-prime to a list of other numbers, and if it is indeed co-prime to all of the input numbers, add n to the state list:

```
>>> def gather_primes(n: int, ls: list[int]) -> tuple[bool, list[int]]:
...     if any(n % i == 0 for i in ls):
...         return (False, ls)
...     return (True, ls + [n])
```

Example uses of this follow:

```
>>> gather_primes(2, [])
(True, [2])
>>> gather_primes(3, [2])
(True, [2, 3])
>>> gather_primes(4, [2, 3])
(False, [2, 3])
```

This way, we can use `process` to generate prime numbers and do primality testing!

```
>>> def primes(n: int) -> tuple[list[bool], list[n]]:
...     return process(gather_primes, list(range(2, n)), [])
...
>>> primes(10)
([True, True, False, True, False, True, False, False], [2, 3, 5, 7])
>>> primes(30)
([True, True, False, True, False, True, False, False, False, # 2 to 10
  True, False, True, False, False, False, True, False, True, # 11 to 20
  False, False, False, True, False, False, False, False, False, True],
  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29])
```

Proceed to define the `process` function. Example runs are as above.

Note: The type of a function that receives parameters `A`, `B` and `C` and returns `D` is `Callable[[A, B, C], D]`. You will need to import `Callable` from `typing`.

LAST UPDATED **26 OCT 2024**

Two of the most important aspects of software engineering design are *decoupling* and *extensibility*, reducing the dependencies between two systems or programming constructs and making it easy to extend implementations. These are not simple problems for programming language designers to solve. Different languages offer different solutions to this problem, and some languages make these not-so-easy.

In this chapter, we discuss how Haskell allows us to decouple types and functions, and in some sense, making data types extensible, without compromising on type-safety. Haskell does so with a programming feature not common to many languages, known as typeclasses.

LAST UPDATED **26 OCT 2024**

Ad-Hoc Polymorphism

So far, we have learnt how to define algebraic data types, and construct—and destructure—terms of those types. However, algebraic data types typically only represent data, unlike objects in OOP. Therefore, we frequently write *functions* acting on terms of those types. As an example, drawing from Chapter 2.5 Question 7, let us define a `Shape` ADT that represents circles and rectangles.

```
data Shape = Circle Double
           | Rectangle Double Double
```

On its own, this ADT does not do very much. What we would like to do additionally, is to define a function over `Shape`s. For example, a function `area` that computes the area of a `Shape`:

```
area :: Shape -> Double
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

However, you might notice that `area` should not be exclusively defined on `Shape`s; it could very well be the case that we will later define other algebraic data types from which we can also compute its area. For example, let us define a `House` data type that also has a way to compute its area:

```
data House = H [Room]
type Room = Rectangle

area' :: House -> Double
area' (H ls) = foldr ((+) . area) 0 ls
```

Notice that we cannot, at this point, abstract `area` and `area'` into a single function because these functions work on specific types, and they have type-specific implementations. It is such a waste for us to have to use different names to describe the same idea.

The question then becomes, is it possible for us to define an `area` function that is polymorphic (not fully parametrically polymorphic) in some ad-hoc way? That is, can `area` have one implementation when given an argument of type `Shape`, and another implementation when given another argument of type `House`?

Ad-Hoc Polymorphism in Python

Notice that this is entirely possible in Python and other OO languages, where different classes can define methods of the same name.

```
@dataclass
class Rectangle:
  w: float
  h: float
  def area(self) -> float:
    return self.w * self.h

@dataclass
class Circle:
  r: float
  def area(self) -> float:
    return pi * r ** 2

@dataclass
class House:
  ls: list[Rectangle]
  def area(self) -> float:
    return sum(x.area() for x in self.ls)
```

All of these disparate types can define an `area` method with its own type-specific implementation, and this is known as method *overloading*. In fact, Python allows us to use them in an ad-hoc manner because Python does not enforce types. Therefore, a program like the following will be totally fine.

```
def total_area(ls):
  return sum(x.area() for x in ls)

ls = [Rectangle(1, 2), House([Rectangle(3, 4)])]
print(total_area(ls)) # 14
```

`total_area` works because Python uses *duck typing*—if it walks like a duck, quacks like a duck, it is probably a duck. Therefore, as long as the elements of the input list `ls` defines a method `area` that returns something that can be summed over, no type errors will present from program execution.

Python allows us to take this further by defining special methods to overload operators. For example, we can define the `__add__` method on any class to define how it should behave under the `+` operator:

```
@dataclass
class Fraction:
  num: int
  den: int
  def __add__(self, f):
    num = self.num * f.den + f.num * self.den
    den = self.den * f.den
    return Fraction(num, den)

print(1 + 2) # 3
print(Fraction(1, 2) + Fraction(3, 4)) # Fraction(10, 8)
```

However, relying on duck typing alone forces us to ditch any hopes for static type checking. From the definition of the `ls` variable above:

```
ls = [Rectangle(1, 2), House([Rectangle(3, 4)])]
```

based on what we know, `ls` cannot be given a suitable type that is useful. Great thing is, Python has support for *protocols* that allow us to group classes that adhere to a common interface (without the need for class extension):

```
class HasArea(Protocol):
  @abstractmethod
  def area(self) -> float:
    pass

def total_area(ls: list[HasArea]) -> float:
  return sum(x.area() for x in ls)

ls: list[HasArea] = [Rectangle(1, 2), House([Rectangle(3, 4)])]
print(total_area(ls)) # 14
```

This is great because we have the ability to perform ad-hoc polymorphism without coupling the data with behaviour—the `HasArea` protocol makes no mention of its inhabiting classes `Rectangle`, `Circle` and `House`, and vice-versa, and yet we have provided enough information for the type-checker so that bogus code such as the following gets flagged early.

```
ls: list[HasArea] = [1] # Type checker complains about this
print(total_area(ls)) # TypeError
```

The Expression Problem in Python

There are several limitations of our solution using protocols. Firstly, Python's type system is not powerful or expressive enough to describe protocols involving higher-kinded types. Secondly, although we have earlier achieved decoupling between classes and the protocols they abide by, we are not able to decouple classes and their methods. If we wanted to

completely decouple them, we would define methods as plain functions, and run into the same problems we have seen in the Haskell implementation of `area` and `area'` above.

At the expense of type safety, let us attempt to decouple `area` and their implementing classes. The idea is to define an `area` function that receives a helper function that computes the type specific area of an object:

```
def area(x, helper) -> float:
  return helper(x)

def rectangle_area(rect: Rectangle) -> float:
  return rect.w * rect.h
def house_area(house: House) -> float:
  return sum(x.area() for x in house.ls)

r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r, rectangle_area) # 2
area(h, house_area) # 12
```

This implementation is silly because we could easily remove one level of indirection by invoking `rectangle_area` or `house_area` directly. However, notice that the implementations are specific to classes—or, types—thus, what we can do is to store these helpers in a dictionary whose keys are the types they were meant to be inhabiting. Then, the `area` function can look up the right type-specific implementation based on the type of the argument.

```
HasArea = {}

def area(x):
  helper = HasArea[type(x)]
  return helper(x)

HasArea[Rectangle] = lambda x: x.w * x.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)

r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

What's great about this approach is that (1) otherwise disparate classes adhere to a common interface, and (2) the classes and methods are completely decoupled. We can later on define additional classes and its type-specific implementation of `area`, or define a type-specific implementation of `area` for a class that has already been defined!

```python
@dataclass
class Triangle:
  w: float
  h: float

HasArea[Triangle] = lambda t: 0.5 * t.w * t.h

area(Triangle(5, 2)) # 5
```

Unfortunately, all of these gains came at the cost of type safety. Is there a better way to do this? In Haskell, yes—with typeclasses!

LAST UPDATED **26 OCT 2024**

Typeclasses

Typeclasses are a type system construct that enables ad-hoc polymorphism. Essentially, a typeclass is a nominal classification of types that all support some specified behaviour, by having each type providing its type-specific implementation for those behaviours. Alternatively, a typeclass can be seen as a constraint for a type to support specified behaviours.

Just like classes in OOP are blueprints for creating instances of the class (objects), a typeclass is a blueprint for creating typeclass instances. This time, a typeclass provides the interface/specification/contract for members of the typeclass to adhere to, and typeclass instances provide the actual type-specific implementations of functions specified in the typeclass. In essence, a typeclass is a constraint over types, and a typeclass instance is a witness that for types meeting those constraints.

To build on intuition, pretend that there is a super cool magic club, and members of this club must have a magic assistant and a magic trick. This club acts as a typeclass. Then suppose cats and dogs want to join this club. To do so, they must provide proof to the club administrators (in Haskell, the compiler) that they have a magic assistant and a magic trick. Suppose that the cats come together with their mouse friends as their magic assistants, and their magic trick is to cough up a furball, and the dogs all present their chew toys as their magic assistants, and their magic trick is to give their paw. The club administrator then puts all these into boxes as certificates of their membership into the club—in our analogy, these certificates are typeclass instances.

Let us return to the shape and house example we have seen at the start of this chapter. We first define some types (slightly different from before) that all have an area:

```
data Shape = Circle Double
           | Rectangle Double Double
           | Triangle Double Double
data House = H [Room]
data room = R { roomName :: String
              , shape    :: Shape }
```

Now, our goal is to describe the phenomenon that some types have an area. For this, we shall describe a contract for such types to follow. The contract is straightforward—all such types must have an `area` function (known as a method).

```
class HasArea a where
  area :: a -> Double
```

An important question one might ask is: why is `HasArea` polymorphic? To give an analogy, recall in our Python implementation with dictionaries that `HasArea` is a dictionary where we are looking up type-specific implementations of `area` by type. Essentially, it is a finite map or (partial) function from types to functions. This essentially makes `HasArea` polymorphic, because it acts as a function that produces different implementations depending on the type!

Then, the `area` function should also receive a parameter of type `a` —that is, if `a` is a member of the `HasArea` typeclass, then there is a function `area :: a -> Double`. The example typeclass instances make this clear:

```
instance HasArea Shape where
  area :: Shape -> Double
  area (Circle r) = pi * r ^ 2
  area (Rectangle w h) = w * h
  area (Triangle w h) = w * h / 2

instance HasArea Room where
  area :: Room -> Double
  area x = area $ shape x

instance HasArea House where
  area :: House -> Double
  area (H rooms) = sum $ map area rooms
```

Each instance of `HasArea` provides a type-specific implementation of `area`. For example, the `HasArea Shape` instance acts as a witness that `Shape` belongs to the `HasArea` typeclass. It does so by providing an implementation of `area :: Shape -> Double` (in the obvious way). We do the same for rooms and houses, and now the `area` function works for all (and only) these three types!

```
x :: Shape = Triangle 2 3
y :: Room = R "bedroom" (Rectangle 3 4)
z :: House = H [y]

ax = area x -- 3
ay = area y -- 12
az = area z -- 12
```

Now let us investigate the type of `area`:

```
ghci> :t area
area :: forall a. HasArea a => a -> double
```

The type of `area` is read as "a function for all `a` where `a` is constrained by `HasArea`, and receives an `a`, and returns a `Double`".

Constrains on type variables are not limited to class methods. In fact, we can, and probably should, make functions that use `area` polymorphically over type variables, constrained by

`HasArea`. Let us consider a function that sums the area over a list of shapes, and another one over a list of rooms:

```
totalArea :: [Shape] -> Double
totalArea [] = 0
totalArea (x : xs) = area x + totalArea xs

-- alternatively
totalArea' :: [Shape] -> Double
totalArea' = sum . map area

totalArea'' :: [Room] -> Double
totalArea'' = sum . map area
```

Both `totalArea'` and `totalArea''` have precisely the same implementation, except that they operate over `Shape` and `Room` respectively. We can substitute these types for any type variable `a`, so long as there is an instance of `HasArea a`! Therefore, the most general type we should ascribe for this function would be

```
totalArea :: HasArea a => [a] -> Double
totalArea = sum . map area
```

Now our `totalArea` function works on any list that contains a type that has an instance of `HasArea`!

```
xs :: [Shape] = [Rectangle 1 2, Triangle 3 4]
ys :: [House] = [H [R "bedroom" (Rectangle 1 2)]]
axs = totalArea xs -- 8
ayx = totalArea ys -- 2
```

How Typeclasses Work

By now, you should be able to observe that typeclasses allow (1) otherwise disparate types adhering to a common interface, i.e. ad-hoc polymorphism and (2) decoupling types and behaviour, all in a type-safe way—this is very difficult (if not impossible) to achieve in other languages like Python. The question then becomes: how does Haskell do it?

The core idea behind typeclasses and typeclass instances is that typeclasses are implemented as regular algebraic data types, and typeclass instances are implemented as regular terms of typeclasses. Using our `area` example, we can define the typeclass as

```
data HasArea a = HA { area :: a -> Double }
```

Then, typeclass instances are merely helper-terms of the `HasArea` type:

```haskell
hasAreaShape :: HasArea Shape
hasAreaShape = HA $ \x -> case x of
  Circle    r   -> pi * r ^ 2
  Rectangle w h -> w * h
  Triangle  w h -> w * h / 2
```

Notice that `area` now has the type `HasArea a -> a -> Double`. Clearly, `area hasAreaShape` is now the `Shape`-specific implementation for obtaining the area of a shape! We can take this further by defining the helper-terms for other types that wish to implement the `HasArea` typeclass:

```haskell
hasAreaRoom :: HasArea Room
hasAreaRoom = HA $ \x -> area hasAreaShape (shape x)

hasAreaHouse :: HasArea House
hasAreaHouse = HA $ \x -> case x of
  H rooms -> sum $ map (area hasAreaRoom) rooms
```

Finally, we can use the `area` function, together with the type-specific helpers, to compute the area of shapes, rooms and houses!

```haskell
x :: Shape = Triangle 2 3
y :: Room = R "bedroom" (Rectangle 3 4)
z :: House = H [y]

ax = area hasAreaShape x -- 3
ay = area hasAreaRoom y -- 12
az = area hasAreamHouse z -- 12
```

This is (more-or-less) how Haskell implements typeclasses and typeclass instances. The only difference is that the Haskell compiler will automatically *infer* the helper term when a typeclass method is used, allowing us to omit them. This *term inference* that Haskell supports allow us to define and use ad-hoc polymorphic functions in a type-safe way.

LAST UPDATED 26 OCT 2024

Commonly Used Typeclasses

Let us have a look at some typeclasses and their methods that you have already used.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer a
```

Equality Comparisons

The `Eq` typeclass describes types that are amenable to equality comparisons; the `Num` typeclass describes types that can behave as numbers, with support for typical numeric operations like addition, subtraction and so on. Haskell's Prelude already ships with the instances of these typeclasses for commonly-used types, such as instances for `Num Int` and `Eq String`.

Let us try defining our own instance of `Eq`. Suppose we are re-using the `Fraction` algebraic data type defined in Chapter 2.3 (Types#Algebraic Data Types):

```
data Fraction = Fraction Int Int
```

We allow `Fraction` to be amenable to equality comparisons by implementing a typeclass instance for `Eq Fraction`:

```
instance Eq Fraction where
  (==) :: Fraction -> Fraction -> Bool
  F a b == F c d = a == c && b == d

  (/=) :: Fraction -> Fraction -> Bool
  F a b /= F c d = a /= c || b /= d
```

Firstly, notice that we are performing equality comparisons between the numerators and denominators. This is okay because we know that the numerators and denominators of

fractions are integers, and there is already an instance of `Eq Int`. Next, usually by definition, `a /= b` is the same as `not (a == b)`. Therefore, having to always define both `(==)` and `(/=)` for every instance is cumbersome.

Minimal Instance Definitions

Let us inspect the definition of the `Eq` typeclass:

```
ghci> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
    -- Defined in 'GHC.Classes'
```

Notice the `MINIMAL` *pragma*—the pragma states that we only need to define either `(==)` *or* `(/=)` for a complete instance definition! Therefore, we can omit the definition of `(/=)` in our `Eq Fraction` instance, and we would still have a complete definition:

```
instance Eq Fraction where
  (==) :: Fraction -> Fraction -> Bool
  F a b == F c d = a == c && b == d


ghci> Fraction 1 2 == Fraction 1 2
True
ghci> Fraction 1 2 /= Fraction 1 2
False
```

A natural question to ask is, why not simply define `Eq` to only have `(==)` and give `(/=)` for free?

```
class Eq a where
  (==) :: a -> a -> Bool

(/=) :: Eq a => a -> a -> Bool
x /= y = not (x == y)
```

By placing both functions as methods in the typeclass, programmers have the option to define *either* `(==)` *or* `(/=)`, or both, if specifying each implementation individually gives better performance or different behaviour than the default.

Typeclass Constraints in Typeclasses and Instances

We can even define instances over polymorphic types. Here is an example of how we can perform equality comparisons over trees:

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

instance Eq (Tree a) where
  (==) :: Tree a -> Tree a -> Bool
  Empty == Empty = True
  (Node l v r) == (Node l' v' r') = l == l' && v == v' && r == r'
  _ == _ = False
```

However, our instance will not type-check because the elements `a` of the trees also need to be amenable to equality comparisons for us to compare trees! Therefore, we should constrain `a` with `Eq` in the *instance* declaration, like so:

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

instance Eq a => Eq (Tree a) where
  (==) :: Tree a -> Tree a -> Bool
  Empty == Empty = True
  (Node l v r) == (Node l' v' r') = l == l' && v == v' && r == r'
  _ == _ = False
```

In fact, we can write typeclass constraints in typeclass declarations as well. For example, the `Ord` typeclass describes (total) orders on types, and all (totally) ordered types must also be amenable to equality comparisons:

```
class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  -- ...
```

Deriving Typeclasses

In fact, some typeclasses are so straightforward that defining instances of these classes are a tedium. For example, the `Eq` class is (usually) very straightforward to define—two terms are equal if they are built with the same constructor and their argument terms are respectively equal. As such, the language should not require programmers to implement straightforward instances of classes like `Eq`.

Haskell has a *deriving mechanism* that allows the compiler to automatically synthesize typeclass instances for us. It is able to do so for `Eq`, `Ord`, and others like `Enum`. Doing so is

incredibly straightforward:

```
data A = B | C

data Fraction = Fraction Int Int
  deriving Eq -- deriving Eq Fraction instance

data Tree a = Empty | Node (Tree a) a (Tree a)
  deriving (Eq, Show) -- deriving Eq (Tree a) and Show (Tree a)

deriving instance Eq A -- stand-alone deriving declaration
```

These declarations tell the compiler to synthesize instance declarations in the most obvious way. This way, we do not have to write our own instance declarations for these typeclasses!

```
ghci> x = Node Empty 1 Empty
ghci> y = Node (Node Empty 1 Empty) 2 Empty
ghci> x
Node Empty 1 Empty
ghci> x == y
False
```

LAST UPDATED **26 OCT 2024**

Functional Dependencies

Observe the type of `(+)`:

```
:t (+)
(+) :: forall a. Num a => a -> a
```

This is quite different in Python:

```
>>> type(1 + 1)
class <'int'>
>>> type(1 + 1.0)
class <'float'>
>>> type(1.0 + 1)
class <'float'>
>>> type(1.0 + 1.0)
class <'float'>
```

The `+` operator in Python behaves heterogenously—when given two `int`s we get an `int`; when given at least one `float` we get a `float`. How would we encode this in Haskell?

Simple! Create a multi-parameter typeclass that describes the argument types and the result type!

```
class (Num a, Num b, Num c) => HAdd a b c where
  (+#) :: a -> b -> c
```

Then we can write instances for the possible permutations of the desired types:

```
instance Num a => HAdd a a a where
  (+#) :: a -> a -> a
  (+#) = (+)

instance HAdd Int Double Double where
  (+#) :: Int -> Double -> Double
  x +# y = fromIntegral x + y

instance HAdd Double Int Double where
  (+#) :: Double -> Int -> Double
  x +# y = x + fromIntegral y
```

However, trying to use `(+#)` is very cumbersome:

```
ghci> x :: Int = 1
ghci> y :: Double = 2.0
ghci> x +# y
<interactive>:3:1: error:
    - No instance for (HAdd Int Double ()) arising from a use of 'it'
    - In the first argument of 'print', namely 'it'
      In a stmt of an interactive GHCi command: print it
ghci> x +# y :: Double
3.0
```

This occurs because without specifying the return type `c`, Haskell has no idea what it is since it is ambiguous! As per the definition, no one is stopping us from defining another `instance HAdd Int Double String`! On the other hand, we know that adding an `Int` and a `Double` *must* result in a `Double` and nothing else; in other words, the types of the arguments to `(+#)` *uniquely characterizes* the resulting type.

The way we introduce this dependency between these type variables by introducing *functional dependencies* on typeclass declarations, which, adding them to our declaration of `HAdd`, looks something like the following:

```
{-# LANGUAGE FunctionalDependencies #-}
class (Num a, Num b, Num c) => HAdd a b c | a b -> c where
  (+#) :: a -> b -> c
```

The way to read the clause `a b -> c` is "`a` and `b` *uniquely characterizes/determines* `c`", or in other words, `c` is a *function* of `a` and `b`, i.e. it is not possible that given a *fixed* `a` and `b` that we have two different inhabitants of `c`. This (1) prevents the programmer from introducing different values of `c` for the same `a` and `b` (which we haven't) and (2) allows the compiler to infer the right instance just with `a` and `b` alone.

```
ghci> x :: Int = 1
ghci> y :: Double = 2.0
ghci> x +# y
3.0
ghci> :{
ghci| instance HAdd Int Double String where
ghci|   x +# y = show x
ghci| :}
<interactive>:8:10: error:
    Functional dependencies conflict between instance declarations:
      instance [safe] HAdd Int Double Double
        -- Defined at <interactive>:17:10
      instance HAdd Int Double String -- Defined at <interactive>:21:10
```

LAST UPDATED **26 OCT 2024**

The Existential Typeclass Antipattern

In Python, as long as a class abides by a protocol, the Python type system presumes that this class is a *subclass* of said protocol. Therefore, any object instantiated from such a class is also considered to be of the same type as the protocol. Thus, in our earlier example, shapes, houses and rooms are all considered to be the same type has `HasArea`.

```python
class HasArea(Protocol):
    def area(self) -> float:
        pass

@dataclass
class Rectangle:
    # ...
    def area(self) -> float:
        return # ...

@dataclass
class House:
    # ...
    def area(self) -> float:
        return # ...

# the following is ok and well-typed
ls: list[HasArea] = [Rectangle(1, 2), House(...)]
```

However, this is *not* okay in Haskell because `HasArea` is not a type, but a typeclass!

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]
ls = [x, y, z] -- error!
```

One question we might ask is, how do we replicate this ability in Python? I.e., how do we create a type that represents all types that implement `HasArea` in Haskell?

Existential Types

Recall that polymorphic types are also called for-all types. Essentially, the definition of the type is independent of the type parameter. The idea behind for-all types is that we can substitute the type parameter with any other type to give a new type. For example, we know that the `id` function has type `forall a. a -> a`. Therefore, we can apply `id` onto a type, say `Int`, to give us a new function whose type is `Int -> Int`.

The type variable `a` is opaque to whoever defines the term of the polymorphic type. For example, when we define a polymorphic function:

```
singleton :: forall. a -> [a]
singleton x = []
```

The type of `x` is just `a` where we have no idea what `a` is. Thus, the implementation of `singleton` cannot make use of any knowledge of what `a` is because it is just an opaque type variable. In contrast, anyone who *uses* `singleton` can *decide* what type will inhabit `a`:

```
x :: Int
y = singleton @Int x
```

As you can see, the caller of `singleton` can decide to pass in the type `Int`, and thus will know that the function application `singleton @Int x` will evaluate to a term of type `[Int]`.

One question you might ask is, we know that "for all" corresponds to $\forall$ in mathematics. Are there also $\exists$ types? The answer is yes! These are known as *existential types*:

$$\exists \alpha . \tau$$

The idea behind existential types is that there is *some* type which inhabits the existential type variable to give a new type. For example the type $\exists \alpha . [\alpha]$ means *"some"* list of elements. The term `[1, 2]` can also be treated as having the type $\exists \alpha . [\alpha]$ because we know that we can let α be `Int` and `[1, 2]` is correctly of type `[Int]`. Similarly, `"abc"` can also be treated as having the type $\exists \alpha . [\alpha]$ because we know that we can let α be `Char` and `"abc"` is correctly of type `[Char]`. However, `[1, 'a']` is *not* of type $\exists \alpha . [\alpha]$ since we cannot assign any type to α so that the type of `[1, 'a']` matches it.

An existential type reverses the relationship of type variable opacity. Recall that the implementer of a polymorphic function sees the type variable as opaque, while the user gets to decide what type inhabits the type variable. For an existential type, the implementer gets to decide what type inhabits the type variable, while the user of an existential type views the type variable as opaque.

> **Polymorphism**: implementer **does not know the type, must ignore it**. **User** chooses the type.
>
> **Existential types**: implementer **chooses the type**. **User** does not know the type, must ignore it.

Ideally, this allows us to define a type of lists $[\exists \alpha . \texttt{HasArea}\ \alpha \Rightarrow \alpha]$ (read: a list of elements, each of which are some α that implements `HasArea`), however the quantification of the type variable is inside the list constructor; these are called *impredicative* types. Haskell does not support impredicative types. What can we do now?

What we can try to do is to define a new wrapper type that stores elements of type $\exists \alpha.\, \texttt{HasArea}\ \alpha$, like so:

```
data HasAreaType = HAT (∃a. HasArea a => a)
instance HasArea HasAreaType where
    area :: HasAreaType -> Double
    area (HAT x) = area x
```

However, perhaps surprisingly given what we've been talking about, Haskell does not even support existential types. What now?

Mental Model for Existential Types

Just like how we have given a mental model for polymorphism, we give a mental model for existential types. Recall that a polymorphic function is a function that receives a type parameter and returns a function that is specialized over the type parameter. For us, let us suppose that a term of an existential type $\exists \alpha.\, \tau$ is a pair (β, x) such that x has type $\tau[\alpha := \beta]$.

- `(Int, [1, 2])` is a term of type $\exists \alpha.\, [\alpha]$ because `[1, 2] :: [Int]`
- `(Char, "abc")` is also a term of type $\exists \alpha.\, [\alpha]$ because `"abc" :: [Char]`

Therefore, a function on an existential type can be thought of as a function receiving a pair, whose first element is a type, and the second element is corresponding term.

In our example above, the `HAT` constructor would therefore have type

```
HAT :: (∃a. HasArea a => a) -> HasAreaType
```

Using our mental model, we destructure the existential type as a pair:

```
HAT :: (a :: *, HasArea a => a) -> HasAreaType
```

Recall *currying*, where a function over more than one argument is split into a function receiving one parameter and returning a function that receives the rest. We thus curry the `HAT` constructor like so:

```
HAT :: (a :: *) -> HasArea a => a -> HasAreaType
```

Remember what it means for a function that receives a type as a parameter—this is a **polymorphic function**!

```
HAT :: forall a. HasArea a => a -> HasAreaType
```

Indeed, *polymorphic functions simulate functions over existential types*. Let us show more examples of this being the case. For example, the `area` typeclass method is a function over *something that implements* `HasArea`, and returns a `Double`. Therefore, it should have the following function signature:

```
area :: (∃a. HasArea a => a) -> Double
```

However, we know that we can curry the existential type to get a polymorphic function, allowing us to recover the original type signature!

```
area :: forall a. HasArea a => Double
```

In another example, we know the `EqExpr` constructor from the previous chapter is constructed by providing any two expressions that are amenable to equality comparisons:

```
EqExpr :: (∃a. Eq a => (Expr a, Expr a)) -> Expr Bool
```

Again, with currying, we recover our original type signature for `EqExpr`:

```
EqExpr :: forall a. Eq a => Expr a -> Expr a -> Expr Bool
```

With this in mind, we can now properly create our `HAT` constructor and use the `HasAreaType` type to put shapes, rooms and houses in a single list!

```
data HasAreaType where
    HAT :: forall a. HasArea a => a -> HasAreaType
instance HasArea HasAreaType where
    area :: HasAreaType -> area
    area (HAT x) = area x

x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]

ls :: [HasAreaType]
ls = [HAT x, HAT y, HAT z]

d = totalArea ls -- 27
```

The Antipattern

Notice that we went through this entire journey just so that we can put these different types in a list, which is so that we can compute the total area. However, in this case, we can actually just save the trouble and do this:

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]

ls :: [Double]
ls = [area x, area y, area z]

d = sum ls -- 27
```

Of course, there are definitely use cases for existential types like `HasAreaType`. We frequently call these *abstract data types*. However, these are not commonly used. In fact, not knowing what existential types are should **not** affect your understanding of type classes and polymorphic types. In addition, encoding existential types as pairs is *very handwave-y* and is not even supported in Haskell. The closest analogue of real-world existential types is *dependent pair types* or Σ*-types*, which is different to the existential types we have seen. The demonstration that we have seen so far only serves as a mental model for why we write polymorphic functions were the return type does not depend on the type parameters.

The key point is that we should *not* immediately attempt to replicate OO design patterns in FP just because they are familiar. Trying to skirt around the restrictions of the type system is, generally, not a good idea (there are cases where that is useful, but such scenarios occur exceedingly infrequently).

LAST UPDATED **26 OCT 2024**

Exercises

Question 1

Without using GHCI, determine the types of the following expressions:

1. `1 + 2 * 3`
2. `(show . )`
3. `( . show)`
4. `\ (a, b) -> a == b`

Question 2

You are given the following untyped program:

```
type Tree[a] = Empty | TreeNode[a]
type List[a] = Empty | ListNode[a]

@dataclass
class Empty:
    def to_list(self):
        return []

@dataclass
class ListNode[a]:
    head: a
    tail: List[a]
    def to_list(self):
        return [self.head] + self.tail.to_list()

@dataclass
class TreeNode[a]:
    l: Tree[a]
    v: a
    r: Tree[a]
    def to_list(self):
      return self.l.to_list() + [self.v] + self.r.to_list()

def flatten(ls):
    if not ls: return []
    return ls[0].to_list() + flatten(ls[1:])

ls = [ListNode(1, Empty()), TreeNode(Empty(), 2, Empty())]
ls2 = flatten(ls)
```

Fill in the type signatures of all the methods and functions and the type annotations for the `ls` and `ls2` variables so that the type-checker can verify that the program is type-safe. The given type annotations should be general enough such that defining a new class and adding an instance of it to `ls` requires no change in type annotation:

```
@dataclass
class Singleton[a]:
  x: a
  def to_list(self):
    return [self.x]

ls = [ListNode(1, Empty()), TreeNode(Empty(), 2, Empty()),
  Singleton(3)]
# ...
```

Question 3

Defined below is a data type describing clothing sizes.

```
data Size = XS | S | M | L | XL
    deriving (Eq, Ord, Show, Bounded, Enum)
```

Proceed to define the following functions:

- `smallest` produces the smallest size
- `descending` produces a list of all the sizes from large to small
- `average` produces the average size of two sizes; in case there isn't an exact middle between two sizes, prefer the smaller one

Example runs follow.

```
ghci> smallest :: Size
XS
ghci> descending :: [Size]
[XL, L, M, S, XS]
ghci> average XS L
S
```

However, take note that your functions must *not* only work on the `Size` type. Some of these functions can be implemented with the typeclass methods that `Size` derives. You should implement your solution based on these methods so that your function can be as general as possible. In particular, we should be able to define a new type which derives these typeclasses, and all your functions should still work on them as we should expect. An example is as follows:

```
ghci> :{
ghci| data Electromagnet = Radio | Micro | IR | Visible | UV | X | Gamma
ghci|    deriving (Eq, Ord, Show, Bounded, Enum)
ghci| :}
ghci> smallest :: Electromagnet
Radio
ghci> descending :: [Electromagnet]
[Gamma, X, UV, Visible, IR, Micro, Radio]
ghci> average Gamma Radio
Visible
```

Question 4

Implement the mergesort algorithm as a function `mergesort`. Ignoring time complexity, your algorithm should split the list in two, recursively mergesort each half, and merge the two sorted sublists together. Example runs follow:

```
ghci> mergesort [5,2,3,1,2]
[1,2,2,3,5]
ghci> mergesort "edcba"
"abcde"
```

Question 5

Recall Chapter 2.3 (Types#Algebraic Data Types) where we defined an `Expr` GADT.

```
data Expr a where
  LitNumExpr :: Int -> Expr Int
  AddExpr :: Expr Int -> Expr Int -> Expr Int
  -- ...

eval :: Expr a -> a
eval (LitNumExpr x) = x
eval (AddExpr e1 e2) = eval e1 + eval e2
  -- ...
```

Now that we have learnt typeclasses, let us attempt to *separate* each constructor of `Expr` as *individual types*, while still preserving functionality; the purpose of this being to keep the `Expr` type modular and extensible:

```
data LitNumExpr = -- ...
data AddExpr = -- ...
```

while still being able to apply `eval` on any of those expressions:

```
-- 2 + 3
ghci> eval (AddExpr (LitNumExpr 2) (LitNumExpr 3))
5
-- if 2 == 1 + 1 then 1 + 2 else 4
ghci> eval (CondExpr
  (EqExpr (LitNumExpr 2)
          (AddExpr (LitNumExpr 1) (LitNumExpr 1)))
  (AddExpr (LitNumExpr 1) (LitNumExpr 2))
  (LitNumExpr 4))
3
```

Proceed to define all these different types of expressions and their corresponding implementations for `eval`:

- `LitNumExpr`. A literal integer, such as `LitNumExpr 3`.
- `AddExpr`. An addition expression in the form of $e_1 + e_2$, such as `AddExpr (LitNumExpr 1) (LitNumExpr 2)` representing $1 + 2$
- `EqExpr`. An equality comparison expression in the form of $e_1 = e_2$, such as `Eq (LitNumExpr 1) (LitNumExpr 2)` representing $1 = 2$
- `CondExpr`. A conditional expression in the form of $\text{if } e \text{ then } e_1 \text{ else } e_2$

Question 6

In Python, a *sequence* is a data structure that has a *length* and a way to obtain elements from it by integer indexing. Strings, ranges, tuples and lists are all sequences in Python:

```
>>> len([1, 2, 3])
3
>>> 'abcd'[3]
'c'
```

Our goal is to create something similar in Haskell. However, instead of loosely defining what a *sequence* is, like Python does, we shall create a typeclass called `Sequence` and allow all types that implements these methods to become a sequence formally (at least, to the compiler)!

Proceed to define a typeclass called `Sequence` with two methods:

- `(@)` does indexing, so `ls @ i` is just like `ls[i]` in Python; if the index `i` is out of bounds, the method should panic (you can let it return `undefined` in this case)
- `len` produces the length of the sequence
- `prepend` prepends an element onto the sequence

Then define instances for `[a]` to be a sequence over `a`'s! Example runs follow:

```
ghci> x :: [Int] = [1, 2, 3, 4]
ghci> x @ 2
3
ghci> x @ 4
-- some error...
ghci> len x
4
ghci> x `prepend` 5
[5, 1, 2, 3, 4]
ghci> len "abcde"
5
ghci> "abcde" @ 0
'a'
```

What's really neat about using typeclasses instead of defining a separate `Sequence` data type is that *any* type that conforms to the specification in our `Sequence` typeclass can become a valid sequence. For example, one sequence we might want is a sequence of `()` (the unit type, which only has one constructor with no arguments, and terms of this type signify "*nothing significant*", similar to `void` in other languages).[1] Because each element of such a sequence carries no information, instead of creating such a sequence using a list, i.e. a list of type `[()]`, we can instead use `Int` as our sequence!

```
ghci> x :: Int = 4
ghci> x @ 2
()
ghci> x @ 4
-- some error...
ghci> len x
4
ghci> (x `prepend` 5) @ 4
()
```

Proceed to define a typeclass instance for `Int` such that `Int`s are sequences of `()`.

---

[1] This is an extremely contrived example. The main point we are driving home is that we can create very concise implementations of data structures based on domain-specific knowledge.

LAST UPDATED **26 OCT 2024**

Railways

One of the core ideas in FP is *composition*, i.e. that to "do one computation after the other" is to *compose* these computations. In mathematics, function composition is straightforward, given by:

$$(g \circ f)(x) = g(f(x))$$

That is, $g \circ f$ is the function "g *after* f", which applies f onto x, *and then* apply g on the result.

In an ideal world, composing functions is as straightforward as we have described.

```
def add_one(x: int) -> int:
    return x + 1
def double(x: int) -> int:
    return x * 2
def div_three(x: int) -> float:
    return x / 3

print(div_three(double(add_one(4))))
```

However, things are rarely perfect. Let us take the following example of an application containing users, with several data structures to represent them.

First, we describe the `User` and `Email` classes:

```
from dataclasses import dataclass

@dataclass
class Email:
    name: str
    domain: str

@dataclass
class User:
    username: str
    email: Email
    salary: int | float
```

Now, we want to be able to parse user information that is provided as a string. However, note that this parsing may *fail*, therefore we raise exceptions if the input string cannot be parsed as the desired data structure.

```
def parse_email(s: str) -> Email:
    if '@' not in s:
        raise ValueError
    s = s.split('@')
    if len(s) != 2 or '.' not in s[1]:
        raise ValueError
    return Email(s[0], s[1])

def parse_salary(s: str) -> int | float:
    try:
        return int(s)
    except:
        return float(s) # if this fails and raises an exception,
                        # then do not catch it
```

And to use these functions, we have to ensure that every program point that uses them must be wrapped in a `try` and `except` clause:

```
def main():
    n = input('Enter name: ')
    e = input('Enter email: ')
    s = input('Enter salary: ')
    try:
        print(User(n, parse_email(e), parse_salary(s)))
    except:
        print('Some error occurred')
```

As you can see, exceptions are being thrown everywhere. Generally, it is hard to keep track of which functions raise/handle execptions, and also hard to compose exceptional functions! Worse still, if the program is poorly documented (as is the case for our example), no one actually knows that `parse_salary` and `parse_email` will raise exceptions!

There is a better way to do this—by using the *railway pattern*! Let us write the equivalent of the program above with idiomatic Haskell. First, the data structures:

```
data Email = Email { emailUsername :: String
                   , emailDomain   :: String }
  deriving (Eq, Show)

data Salary = SInt Int
            | SDouble Double
  deriving (Eq, Show)

data User = User { username   :: String
                 , userEmail  :: Email
                 , userSalary :: Salary }
  deriving (Eq, Show)
```

Now, some *magic*. No exceptions are raised in any of the following functions (which at this point, might look like *moon runes*):

```haskell
parseEmail :: String -> Maybe Email
parseEmail email = do
    guard $ '@' `elem` email && length e == 2 && '.' `elem` last e
    return $ Email (head e) (last e)
  where e = split '@' email

parseSalary :: String -> Maybe Salary
parseSalary s =
  let si = SInt <$> readMaybe s
      sf = SDouble <$> readMaybe s
  in  si <|> sf
```

And the equivalent of `main` in Haskell is shown below.[1] Although not apparent at this point, we are *guaranteed* that no exceptions will be raised from using `parseEmail` and `parseSalary`.

```haskell
main :: IO ()
main = do
  n <- input "Enter name: "
  e <- input "Enter email: "
  s <- input "Enter salary: "
  let u = User n <$> parseEmail e <*> parseSalary s
  putStrLn $ maybe "Some error occurred" show u
```

How does this work? The core idea behind the railway pattern is that functions are *pure* and *statically-typed*, therefore, all functions must make *explicit* the kind of *effects* it wants to produce. For this reason, any "exceptions" that it could raise must be *explicitly* stated in its type signature by returning the appropriate term whose type represents some *notion of computation*. Then, any other function that uses these functions with notions of computation must *explicitly* handle those notions of computations appropriately.

In this chapter, we describe some of the core facets of the railway pattern:

- What is it?
- What data structures and functions can we use to support this?
- How do we write programs with the railway pattern?

---

[1] Wait... is this an *imperative* program in Haskell?

LAST UPDATED **26 OCT 2024**

Context/Notions of Computation

Many popular languages lie to you in many ways. An example is what we have seen earlier, where Python functions do not document exceptions in its type signature, and must be separately annotated as a docstrong to denote as such. This is not including the fact that Python type annotations are not enforced at all.

```python
def happy(x: int) -> int:
    raise Exception("sad!")
```

This is not unique to dynamically-typed languages like Python. This is also the case in Java. In Java, checked exceptions must be explicitly reported in a method signature. However, *unchecked exceptions*, as named, do not need to be reported and are not checked by the Java compiler. That is not to mention other possible "lies", for example, it is possible to return nothing (`null`) even if the method's type signature requires it to return "*something*":

```java
class A {
    String something() {
        return null;
    }
}
```

We can't lie in Haskell. In the first place, we *shouldn't* lie in general. What now?

Instead, what we can do is to create the right data structures that represent what is *actually* returned by each function! In the Python example `happy` , what we really wanted to return was *either* an `int` , *or* an exception. Let us create a data structure that represents this:

```haskell
data Either a b = Left a  -- sad path
                | Right b -- happy path
```

Furthermore, instead of returning `null` like in Java, we can create a data structure that represents *either* something, *or* nothing:

```haskell
data Maybe a = Just a  -- happy path
             | Nothing -- sad path
```

This allows the `happy` and `something` functions to be written safely in Haskell as:

```haskell
happy :: Either String Int
happy = Left "sad!"

something :: Maybe String
something = Nothing
```

The `Maybe` and `Either` types act as *contexts* or *notions of computation*:

- `Maybe a` —an `a` or nothing
- `Either a b` —either `a` or `b`
- `[a]` —a list of possible `a`s (nondeterminism)
- `IO a` —an I/O action resulting in `a`

These types allow us to accurately describe what our functions are actually doing! Furthermore, these types "wrap" around a type, i.e. For instance, `Maybe`, `Either a` (for a fixed `a`), `[]` and `IO` all have kind `* -> *`, and essentially provide some *context* around a type.

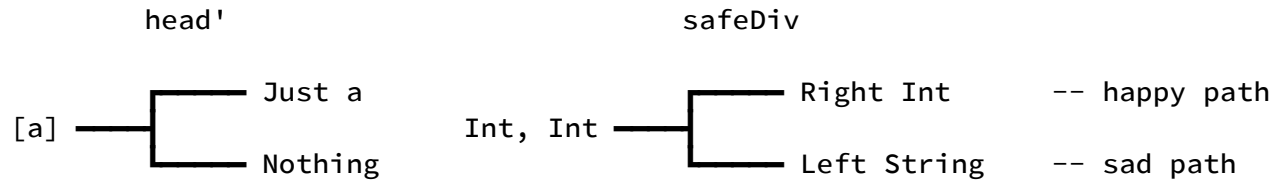
Using these types makes programs clearer! For example, we can use `Maybe` to more accurately describe the `head` function, which may return nothing if the input list is empty.

```haskell
head' :: [a] -> Maybe a
head' [] = Nothing
head' (x : _) = x
```

Alternatively, we can express the fact that dividing by zero should yield an error:

```haskell
safeDiv :: Int -> Int -> Either String Int
safeDiv x 0 = Left "Cannot divide by zero!"
safediv x y = Right $ x `div` y
```

These data structures allow our functions to act as branching railways!

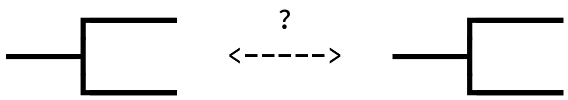


This is the inspiration behind the name "railway pattern", which is the pattern of using algebraic data types to describe the different possible outputs from a function! This is, in fact, a **natural consequence** of purely functional programming. Since functions must be pure, it is not possible to define functions that opaquely cause side-effects. Instead, function signatures must be made *transparent* by using the right data structures.

What, then, is the *right* data structure to use? It all depends on the notion of computation that you want to express! If you want to produce nothing in some scenarios, use `Maybe`. If

you want to produce something or something else (like an error), use `Either`, so on and so forth!

However, notice that having *functions as railways* is not very convenient... with the non-railway (and therefore potentially exceptional) `head` function, we could compose `head` with itself, i.e. `head . head :: [[a]] -> a` is perfectly valid. However, we *cannot* compose `head'` with itself, since `head'` returns a `Maybe a`, which cannot be an argument to `head'`.



How can we make the railway pattern *ergonomic* enough for us to want to use them?

LAST UPDATED **26 OCT 2024**

Category Theory

We can borrow some ideas from a branch of mathematics, known as *Category Theory*, to improve the ergonomics of these structures. Part of the reason why we are able to do so is that all the types that we have described have kind `* -> *`, i.e. they "wrap" around another type. As such, they should be able to behave as *functors*, which we will formalize shortly.[1]

However, before we even talk about what a functor is and how the data structures we have described are functors, we first need to describe what category theory is. Intuitively, most theories (especially the algebraic ones) study mathematical structures that abstract over things; *groups* are abstractions of *symmetries*, and *geometric spaces* are abstractions of *space*. Category theory takes things one step further and studies *abstraction itself*.

Effectively the goal of category theory is to observe similar underlying structures between collections of mathematical structures. What is nice about this is that a result from category theory generalizes to all other theories that fit the structure of a category. As such it should be no surprise that computation can be, and is, studied through the lens of category theory too!

On the other hand, the generality of category theory also makes it incredibly abstract and difficult to understand—this is indeed the case in our very first definition. As such, I will, as much as possible, show you "concrete" examples of each definition and reason about them if I can. With this in mind, let us start with the definition of a category, as seen in many sources.

Definition (Category). A category $\square$ consists of

- a collection of *objects* $\mathrm{X}, \mathrm{Y}, \mathrm{Z}, \ldots$ denoted $\mathrm{ob}(\square)$
- a collection of *morphisms*, $\mathrm{f}, \mathrm{g}, \mathrm{h}, \ldots$, denoted $\mathrm{mor}(\square)$

so that:

- Each morphism has specified *domain* and *codomain* objects; when we write $\mathrm{f} : \mathrm{X} \to \mathrm{Y}$, we mean that the morphism f has domain X and codomain Y.
- Each object has an *identity morphism* $1_{\mathrm{X}} : \mathrm{X} \to \mathrm{X}$.
- For any pair of morphisms f, g with the codomain of f equal to the domain of g (i.e. f and g are composable), there exists a *composite morphism* $\mathrm{g} \circ \mathrm{f}$ whose domain is equal to the domain of f and whose codomain is equal to the codomain of g, i.e.

$$\mathrm{f} : \mathrm{X} \to \mathrm{Y}, \quad \mathrm{g} : \mathrm{Y} \to \mathrm{Z} \quad \rightsquigarrow \quad \mathrm{g} \circ \mathrm{f} : \mathrm{X} \to \mathrm{Z}$$

Composition of morphisms is subject to the two following axioms:

- *Unity*. For any $f : X \to Y$, $f \circ 1_X = 1_Y \circ f = f$.
- *Associativity*. For any composable f, g and h, $(h \circ g) \circ f = h \circ (g \circ f)$.

This, of course, is incredibly abstract and quite hard to take in. Instead, let us use a simpler definition to get some "ideas" across:

A category $\square$ consists of

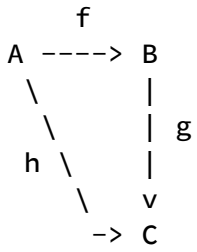
- Dots X, Y, Z
- Arrows between dots $f, g, h, \ldots$

such that:

- Joining two arrows together gives another arrow
- There is a unique way to join three arrows together
- Every dot has an arrow pointing to itself, such that joining it with any other arrow f just gives f

---

Here is an example category:



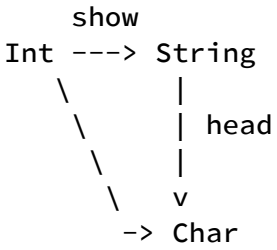
Here we have three objects `A` `B` and `C`, and the morphisms `f: A -> B`, `g: B -> C` and `h: A -> C`. The identity morphisms for the objects are omitted for simplicity. Note that the composition of `f` and `g` exists in the category (assume in the example `g . f == h`).

Why do we care? Well, it turns out that types and functions in Haskell assemble into a category $\square$![2]

- Objects in $\square$ are types like `Int`, `String` etc.
- Morphisms in $\square$ are functions like `(+1)` and `head`

Furthermore,

- The composition of two functions with `(.)` is also a function
- Every type has the identity function `id x = x`, where for all functions `f`, `id . f = f . id = f`



The above is a fragment of ☐. We can see that `show` is a function from `Int` to `String`, and `head` is a function from `String` to `Char`. In addition, the function `head . show` is a function from `Int` to `Char`! Furthermore, all of these types have the identity function `id` which we omit in the diagram.

Still, who cares?

Because the types in Haskell assemble into categories, let's see if there is anything that category theory has to tell us.

Functors

In mathematics, the relationships between objects are frequently far more interesting than the objects themselves. Of course, we do not just focus on *any* relationship between objects, but of keen interest, the *structure preserving* relationships between them, such as group homomorphisms that preserve group structures, or monotonic functions between preordered sets that preserve ordering. In category theory, *functors* are maps between categories that preserve the structure of the domain category, especially the compositions and identities.

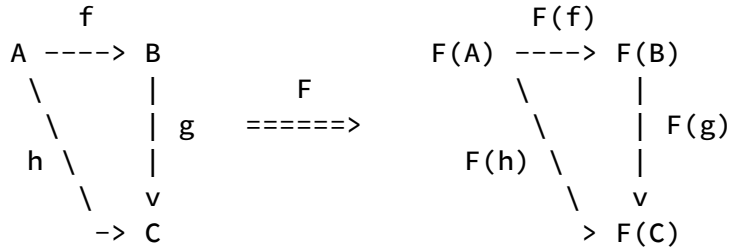
Let ☐ and ☐ be categories. A (*covariant*) *functor* $\mathrm{F} : ☐ \to ☐$ consists of:

- An object $\mathrm{F}(\mathrm{C}) \in \mathrm{ob}(☐)$ for each object $\mathrm{C} \in \mathrm{ob}(☐)$[3].
- A morphism $\mathrm{F}(\mathrm{f}) : \mathrm{F}(\mathrm{C}) \to \mathrm{F}(\mathrm{D}) \in \mathrm{mor}(☐)$ for each morphism $\mathrm{f} : \mathrm{C} \to \mathrm{D} \in \mathrm{mor}(☐)$.

subject to the two *functoriality axioms*:

- For any composable pair of morphisms $\mathrm{f}, \mathrm{g} \in \mathrm{mor}(☐)$, $\mathrm{F}(\mathrm{g}) \circ \mathrm{F}(\mathrm{f}) = \mathrm{F}(\mathrm{g} \circ \mathrm{f})$.
- For each $\mathrm{C} \in \mathrm{ob}(☐)$, $\mathrm{F}(1_{\mathrm{C}}) = 1_{\mathrm{F(C)}}$.

in other words, functors map dots and arrows between two categories, preserving composition and identities.



What's so special about categories and functors, especially since categories are so abstract and have so little requirements for being one? This is precisely the beauty of category theory—it is abstract and simple enough for many things to assemble into one, yet the requirement of associativity and unity of the composition of morphisms and identities make things that assemble into categories behave in the *most obvious way*!

Types as Functors

There are two parts two a functor in ☐:

- Maps types to types
- Maps functions to functions

We already know that the `[]` type constructor maps `a` to `[a]` for all `a` in ☐. How do we map functions `f :: a -> b` to `F(f) :: [a] -> [b]` in the *most obvious way*, i.e. in a way that preserves function composition and identities?

It is simple! Recall the `map` function:

```
>>> def f(x: int) -> str:
...     return str(x + 2)
>>> f(3)
'5'
>>> list(map(f, [3]))
['5']
```

```
ghci> :{
ghci| f :: Int -> String
ghci| f x = show (x + 2)
ghci| :}
ghci> f 3
"5"
ghci> :t map f
map f :: [Int] -> [String]
ghci> map f [3]
["5"]
```

`map` preserves composition:

```
ghci> (map (*2) . map (+3)) [1, 2, 3]
[8, 10, 12]
ghci> map ((*2) . (+3)) [1, 2, 3]
[8, 10, 12]
```

`map` also preserves identities:

```
ghci> :set -XTypeApplications
ghci> map (id @Int) [1, 2, 3]
[1, 2, 3]
ghci> id @[Int] [1, 2, 3]
[1, 2, 3]
```

That is great! `[]` and `map` form a functor over □, which means that we no longer have to worry if someone wants to work in the `[]` context. This is because if we have functions from `a` to `b`, we can *lift* it into a function from `[a]` to `[b]` using `map` and it will behave in the most obvious way!

Can we say the same about `Maybe` and the other type constructors we saw earlier? Fret not! Let's see how we can define a function for `Maybe` so that it can behave as a functor as well! Let's look at `maybeMap`:

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap _ Nothing = Nothing
maybeMap f (Just x) = Just $ f x
```

`maybeMap` also preserves composition and identities!

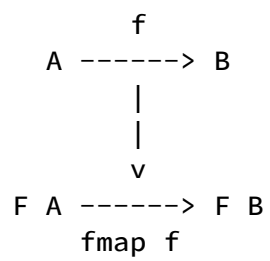
```
ghci> :set -XTypeApplications
ghci> (maybeMap (*2) . maybeMap (+3)) (Just 1)
Just 8
ghci> maybeMap ((*2) . (+3)) (Just 1)
Just 8
ghci> maybeMap (id @Int) (Just 1)
Just 1
ghci> id @(Maybe Int) (Just 1)
Just 1
```

Like we have seen before, all of these types have some `map`-like method that allows us to lift functions into its context; however, they all have their type-specific implementations. This is the reason why Haskell has a `Functor` typeclass!

```haskell
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b]
    fmap _ [] = []
    fmap f (x : xs) = f x : fmap f xs

instance Functor Maybe where
    fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just $ f x

instance Functor (Either a) where -- `a`, a.k.a. sad path is fixed!
    fmap :: (b -> c) -> Either a b -> Either a c
    fmap _ (Left x) = Left x
    fmap f (Right x) = Right $ f x
```

The key point of `[]`, `Maybe`, `Either` etc being functors is as such:

> Given any functor `F` and a function `f` from `A` to `B`, `fmap f` is a function from `F A` to `F B` and **behaves as we should expect**.



Whenever we are presented with a situation that requires us to map a function `f :: A -> B` over a functor `fa :: F A`, just use `fmap f fa` to give us some `fb :: F B`. There is no need to unwrap the `A` from the `F A` (which may not be possible), apply `f` then wrap it back in the `F`; just use `fmap`!

A simple example is as follows. Suppose we have our `head'` function that returns a `Maybe a`, as we have defined earlier. A possible program that we could write that operates on the result of `head'` is the following:

```haskell
ls = [1, 2, 3]
x = head' ls
y = case x of
    Just z -> Just $ z + 1
    Nothing -> Nothing
```

This `case` expression is actually just boilerplate and is not idiomatic! The `Maybe`-specific definition of `fmap` already handles this, therefore, we can re-write this program much more

simply as such:

```
ls = [1, 2, 3]
x = head' ls
y = fmap (+1) x
```

Category Theory and Functional Programming

Although we introduced some formalisms of category theory, rest assured that category theory is **not the main point** of this chapter. Instead, category theory *inspires* tools that support *commonly-used programming patterns* backed by *well-defined theoretical notions*. Therefore, when we say that a type is a functor, not only do we mean that it has an `fmap` definition, we also mean that this definition of `fmap` obeys well-understood laws (in the case of functors, `fmap` preserves compositions and identities) and you can use it assuredly.

That being said, we now have a very powerful tool, `fmap`, that allows us to perform computations in context. What other operations might we need to make the railway pattern more ergonomic?

---

[1] We do not cover category theory in too much detail since it is not *required* for functional programming, although an appreciation of it can help with understanding. For a more detailed walkthrough of the connections between functional programming and category theory, see my article on category theory.

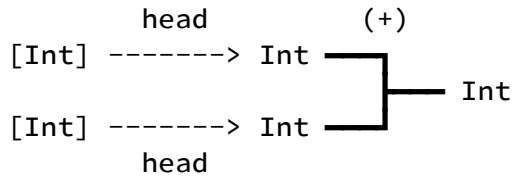
[2] Not really... due to the laziness of Haskell and functions like `seq`, the types and functions in Haskell do not actually assemble in to a category. However, just to put some ideas across, we shall assume that they do.

[3] We abuse the notation of set membership here. It is not necessary for the collections of objects and morphisms of a category to be sets, as is the case for the category of sets.

LAST UPDATED **26 OCT 2024**

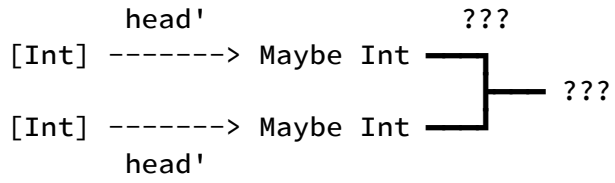
Applicative Functors

What if we had 2 (or more) parallel railways and want to merge them? For example, by using `head`, we can easily retrieve the elements of the list and combine them together in whatever manner we wish:




```haskell
x, y, z :: Int
x = head [1, 2, 3]
y = head [4, 5, 6]
z = x + y -- 5
```

However, when we are using `head'`, combining them is not so easy!




```haskell
x, y :: Maybe Int
x = head' [1, 2, 3]
y = head' [4, 5, 6]
z = x + y -- ???
```

As a first attempt, let us try mapping `(+)` onto `x`:

```haskell
x, y :: Maybe Int
x = head' [1, 2, 3]
y = head' [4, 5, 6]

f :: Maybe (Int -> Int)
f = fmap (+) x
```

The question now is, how do we apply `f :: Maybe (Int -> Int)` above onto `y :: Maybe Int`?

Applicatives

If a functor `f` has the ability to apply `f (a -> b)` onto a `f a` to give an `f b`, then it is an *applicative functor*, which has the same laws of a *(lax-) closed (lax-) monoidal functor* in category theory. Although we could give the formal definition of these, it is quite a lot to unpack, and not necessary for understanding how to use them. Instead, let us directly show the `Applicative` typeclass and some *laws* that govern these typeclass methods.

```haskell
class Functor f => Applicative f where
    -- pure computation in context
    pure :: a -> f a
    -- function application in context
    (<*>) :: f (a -> b) -> f a -> f b
```

These methods are subject to:

- Identity: `pure id <*> v = v`
- Homomorphism: `pure f <*> pure x = pure (f x)`
- Interchange: `u <*> pure y = pure ($ y) <*> u`
- Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

The four laws above, again, govern `Applicatives` to behave in the obvious way. However, as we shall see, there is more than one *obvious way*, therefore, whenever you're using instances of `Functor`s, `Applicative`s and some of the other typeclasses, ensure you read their documentation to understand *which* obvious way it behaves.

Let us look at an example `Applicative` instance:

```haskell
instance Applicative Maybe where
    pure :: a -> Maybe a
    pure = Just

    (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    Nothing <*> _ = Nothing
    _ <*> Nothing = Nothing
    Just f <*> Just x = Just $ f x
```

As you can see, `pure` just raises a value into the `Maybe` context using the `Just` constructor, and `(<*>)` applies a function in context onto an argument in context when they exist. In other words, `pure` and `<*>` behave in the most obvious way.

With this in mind, let us show how we can use `pure` and `<*>` for `Maybe`, but also, applicatives in general. Suppose we have `f :: a -> b -> c`, `x :: a` and `y :: b`. Then, `f x y` would give us something of type `c`.

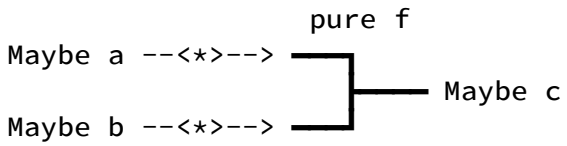
However, Let us raise `x` and `y` into the `Maybe` context, i.e. `x :: Maybe a` and `y :: Maybe b`. Let's see how we can perform the same application (similar to `f x y`) to give us something of `Maybe c`.

To start, we know that we have `<*>` which applies a function in context with an argument in context. Therefore, we first raise `f` into the `Maybe` context using `pure`, then apply it onto `x` using `<*>`:

- `pure f :: Maybe (a -> b -> c)`
- `pure f <*> x :: Maybe (b -> c)`

Finally, using `<*>` again allows us to apply the resulting function onto `y`, giving us a result of type `Maybe c`:

```
pure f <*> x <*> y :: Maybe c
```



However, recall from our very first example that we had attempted to use `fmap` to apply `(+)` onto a `Maybe Int` to give a `Maybe (Int -> Int)`. Now we know that we can directly use this result and apply it onto another `Maybe Int` to give us a `Maybe Int`, thereby applying `(+)` in context! This is a natural consequence of the applicative laws, where `pure f <*> x` is the same as `fmap f x`!

```haskell
pure f <*> x == Just f <*> x
             == case x of
                    Just y -> Just $ f y
                    Nothing -> Nothing
             == fmap f x
```

Therefore, Haskell also defines a function `<$>` as an alias of `fmap`:

```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Therefore, instead of using `pure f <*> x`, we can just write `fmap f x` or `f <$> x` to achieve the same effect!

```
pure f <*> x <*> y = fmap f x <*> y = f <$> x <*> y
```

Now let us revisit our earlier example again! Here is a naive approach to applying `(+)` onto `x` and `y`:

```haskell
x, y, z :: Maybe Int
x = head' [1, 2, 3]
y = head' [4, 5, 6]
z = case (x, y) of
    (Just x, Just y) -> x + y
    _ -> Nothing
```

Don't torture yourself! Instead, knowing that `Maybe` is an applicative (and therefore also a functor), let us just use `<$>` and `<*>`!

```
x, y, z :: Maybe Int
x = head' [1, 2, 3]
y = head' [4, 5, 6]
z = (+) <$> x <*> y -- Just 5
```

As you can see, `Applicative`s allow us to perform computation in context separately, and apply a function over the results over these terms in context!

So far, you should have noticed that the functions and typeclasses presented perform the usual stuff, but in context:

- `fmap :: (a -> b) -> f a -> f b`: lifts a function into a function in context
- `pure :: a -> f a`: puts pure computation in context
- `<*> :: f (a -> b) -> f a -> f b`: function application in context

With these, here are some guidelines for when to use `fmap`, `pure` and `<*>`:

- `f x` becomes `fmap f x` or `f <$> x` or `pure f <*> x` if `x` becomes in context
- `f x` becomes `f <*> x` if both `f` and `x` become in context
- `f x y z` becomes `f <$> x <*> y <*> z` if `x`, `y` and `z` become in context

LAST UPDATED **26 OCT 2024**

Validation

One of the most common use of applicatives is *validation*. From our example at the start of this chapter, we have several data structures and we want to be able to parse them from strings:

```haskell
data Email = Email { emailUsername :: String
                   , emailDomain   :: String }
  deriving (Eq, Show)

data Salary = SInt Int
            | SDouble Double
  deriving (Eq, Show)

data User = User { username   :: String
                 , userEmail  :: Email
                 , userSalary :: Salary }
  deriving (Eq, Show)
```

Parsing them from strings may not always succeed, therefore it is imperative that our parsing function does not guarantee that it returns the desired data structure. Therefore, what we can do instead is to have our parsing functions return results in the `Maybe` context to express this fact. This makes our parsing functions have the following type signatures:

```haskell
parseEmail :: String -> Maybe Email
parseSalary :: String -> Maybe Salary
```

Given these functions, we should be able to define a function that parses a `User` from three strings: the user name (which requires no parsing), the email (which is parsed using `parseEmail`) and the salary (which is parsed using `parseSalary`). One way we can implement this `parseUser` function is by receiving the three strings, performing parsing on the `email` and `salary` (in *parallel*[1]), then constructing our `User` term with the usual `Functor` and `Applicative` methods.

```haskell
parseUser :: String -- name
             -> String -- email
             -> String -- salary
             -> Maybe User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

Now our parsing function works just fine!

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Just (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Nothing
```

Validation with Error Messages

However, this is not always helpful since when parsing a user, several things could go wrong—either (1) the supplied email is invalid, (2) the supplied salary is invalid, or (3) both. Therefore, let's have our parsing functions return an error message instead of `Nothing`. For this, what we want to rely on is the `Either` type, which consists of a `Left` of something sad (like an error message), or a `Right` of something happy (the desired result type). We show the definitions of `Either` and its supporting typeclass instances here.

```
data Either a b = Left a -- sad
                | Right b -- happy

instance Functor (Either a) where
    fmap :: (b -> c) -> Either a b -> Either a c
    fmap _ (Left x) = Left x
    fmap f (Right x) = Right $ f x

instance Applicative (Either a) where
    pure :: b -> Either a b
    pure = Right

    (<*>) :: Either a (b -> c) -> Either a b -> Either a c
    Left f <*> _ = Left f
    _ <*> Left x = Left x
    Right f <*> Right x = Right $ f x
```

Let us change the context that our parsing functions will return. Some of the implementation of `parseEmail` and `parseSalary` will need to be changed to add descriptive error messages, and so will their type signatures.

```
parseEmail :: String -> Either String Email
parseEmail email =
    if ... then
        Left $ "error: " ++ email ++ " is not an email"
    else
        Right $ Email ...

parseSalary :: String -> Either String Salary
parseSalary salary =
    if ... then
        Left $ "error: " ++ salary ++ " is not a number"
    else
        Right $ SInt ...
```

The great thing is that although we have changed the return types of our individual parsing functions, the implementation of `parseUser` does not, because our definition only relies on the typeclass methods of `Functor` and `Applicative`. Since `Either a` is also an `Applicative`, our definition can be unchanged, and only the type signature of `parseUser` needs to be updated.

```
parseUser :: String -- name
             -> String -- email
             -> String -- salary
             -> Either String User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

Now, users of our `parseUser` function will get more descriptive error message reports when parsing fails!

```
ghci> parseUser "Foo" "yong@qi.com" "1000"
Right (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Left "error: yong is not an email"
ghci> parseUser "Foo" "yong@qi.com" "x"
Left "error: x is not a number"
```

Accumulating Error Messages

However, there is one case that is not handled in our validation function. Let's see what that is:

```
ghci> parseUser "Foo" "abc" "x"
Left "error: abc is not an email"
```

Notice that although *both* the email and salaries are invalid, the error message shown *only* highlights the invalid email address. This is misleading because, in fact, the salary is invalid as well, and the user of this function does not know that!

The reason for this lies in the definition of the typeclass instance `Applicative (Either a)`. Notice that in the case of `Left f <*> Left x`, the result is `Left f`, ignoring the other error message `Left x`! In other words, `Either` is a fail-fast `Applicative`, and this is not what we want for our parsing function!

As briefly stated earlier, although the `Applicative` laws describe how an `Applicative` behaves in the *most obvious way*, there is in fact, multiple *most obvious ways* an instance can behave. In fact, we can define a data structure that does not exhibit fail-fastness, and yet, is

still a valid `Applicative` —the result of which is an `Applicative` that allows us to collect all error messages! Let us give this a try.

The first is to re-define `Either` as an ADT called `Validation` that is practically the same (isomorphic) to `Either`, since that structure is still useful for our purposes. The `Functor` instance of this ADT will remain the same.

```
data Validation err a = Success a
                      | Failure err

instance Functor (Validation err) where
    fmap _ (Failure e) = Failure e
    fmap f (Success x) = Success $ f x
```

Notice that our `err` type variable remains as a type variable, instead of a pre-defined error message collection type like `[String]`. This is because, as always, we want to keep our types as general as possible so that it can be used liberally. However, it is now incumbent on us to restrict or constraint `err` in a way that makes it amenable to collecting error messages in an obvious way so that we can still use it for our purposes. In essence, we just need `err` to have some binary operation that is *associative*:

$$\mathrm{E}_1 \oplus (\mathrm{E}_2 \oplus \mathrm{E}_3) = (\mathrm{E}_1 \oplus \mathrm{E}_2) \oplus \mathrm{E}_3$$

For this, we introduce the `Semigroup` typeclass which represents just that!

```
class Semigroup a where
    -- must be associative
    (<>) :: a -> a -> a
```

Any type is a semigroup as long as it is closed under an associative binary operation. With this, as long as our error is a semigroup, we can use that as our errors in `Validation`! Let us define our `Applicative` instance for this:

```
instance Semigroup err => Applicative (Validation err) where
    pure :: a -> Validation err a
    pure = Success

    (<*>) :: Validation err (a -> b) -> Validation err a -> Validation err b
    Failure l <*> Failure r = Failure (l <> r)
    Failure l <*> _ = Failure l
    _ <*> Failure r = Failure r
    Success f <*> Success x = Success (f x)
```

Notice the double-failure case—the errors are combined or aggregated using the semigroup binary operation `(<>)`. This way, no information is lost if both operands are `Failure` cases since they are accumulated together.

Assuredly, using a list of strings as our error log is fine because concatenation is an associative binary operation over lists!

```haskell
instance Semigroup [a] where
    (<>) :: [a] -> [a] -> [a]
    (<>) = (++)
```

Therefore, with these definitions we can now amend our parsing functions to use our new `Validation` `Applicative`. First, as per usual, we amend `parseEmail` and `parseUser` so that they correctly use `Validation` instead of `Either`

```haskell
parseEmail :: String -> Validation [String] Email
parseEmail email =
    if ... then
        Failure ["error: " ++ email ++ " is not an email"]
    else
        Success $ Email ...

parseSalary :: String -> Validation [String] Salary
parseSalary salary =
    if ... then
        Failure ["error: " ++ salary ++ " is not a number"]
    else
        Success $ SInt ...
```

Once again, our `parseUser` function does not need to change, except for the type signature.

```haskell
parseUser :: String -- name
             -> String -- email
             -> String -- salary
             -> Validation [String] User -- user
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s
```

Now, our parsing function works exactly as we want!

```haskell
ghci> parseUser "Foo" "yong@qi.com" "1000"
Success (User "Foo" (Email "yong" "qi.com") 1000)
ghci> parseUser "Foo" "yong" "1000"
Failure ["error: yong is not an email"]
ghci> parseUser "Foo" "yong@qi.com" "x"
Failure ["error: x is not a number"]
ghci> parseUser "Foo" "abc" "x"
Failure ["error: abc is not an email", "error: x is not a number"]
```

Hands-On

In this chapter, we went from parsing with `Maybe` s to parsing with `Either` s and finally to parsing with `Validation` s. Give this a try for yourself!

Written below is the full program for parsing users with `Maybe`. Try replacing the `Maybe`s with `Either`s, then with `Validation`s and see the outcome of running the program each time!

```haskell
module Main where

import Control.Applicative
import Text.Read
import System.IO

-- edit these!
parseEmail :: String -> Maybe Email
parseEmail email =
    if '@' `elem` email && length e == 2 && '.' `elem` last e
    -- edit the following two lines when replacing Maybe with
    -- Either or Validation
    then Just $ Email (head e) (last e)
    else Nothing
  where e = split '@' email

parseSalary :: String -> Maybe Salary
parseSalary s =
  let si = SInt <$> readMaybe s
      sf = SDouble <$> readMaybe s
  in  case si <|> sf of
        Just x -> Just x -- change the RHS `Just x` when replacing
                         -- Maybe with Either or Validation
        Nothing -> Nothing -- change the RHS `Nothing` when replacing
                           -- Maybe with Either or Validation

-- you should only need to change the type of `parseUser` when
-- replacing Maybe with Either or Validation
parseUser :: String -- name
          -> String -- email
          -> String -- salary
          -> Maybe User
parseUser name email salary =
    let e = parseEmail email
        s = parseSalary salary
    in  User name <$> e <*> s

-- no need to edit the rest!

-- the data structures
data Email = Email { emailUsername :: String,
                     emailDomain :: String    }
  deriving (Eq, Show)

data Salary = SInt Int | SDouble Double
  deriving (Eq, Show)

data User = User { username :: String,
                   userEmail :: Email,
                   userSalary :: Salary }
  deriving (Eq, Show)

-- user input with a prompt
input :: String -> IO String
input prompt = do
  putStr prompt
  hFlush stdout
```

```haskell
  getLine

-- splitting strings
split :: Char -> String -> [String]
split _ [] = [""]
split delim (x : xs)
    | x == delim = "" : xs'
    | otherwise  = (x : head xs') : tail xs'
  where xs' = split delim xs

-- validation
data Validation err a = Success a
                      | Failure err
    deriving (Eq, Show)

instance Functor (Validation err) where
    fmap :: (a -> b) -> Validation err a -> Validation err b
    fmap _ (Failure e) = Failure e
    fmap f (Success x) = Success $ f x

instance Semigroup err => Applicative (Validation err) where
    pure :: a -> Validation err a
    pure = Success

    (<*>) :: Validation err (a -> b) -> Validation err a -> Validation err b
    Failure l <*> Failure r = Failure (l <> r)
    Failure l <*> _ = Failure l
    _ <*> Failure r = Failure r
    Success f <*> Success x = Success (f x)

main :: IO ()
main = do
  n <- input "Enter name: "
  e <- input "Enter email: "
  s <- input "Enter salary: "
  print $ parseUser n e s
```

---

[1] It is important to note that the use of the word "parallel" in this chapter has nothing to do with *parallelism*. The word "parallel" is only used to describe the notion of merging parallel railways into a single rail line via `<*>`.

LAST UPDATED **26 OCT 2024**

Monads

Another incredibly useful tool is to be able to perform *composition in context*. That is, that given something of `f a` and a function from `a -> f b`, how do we get an `f b`?

Consider the following example. We can write 123 divided by 4 *and then* divided by 5 via the following straightforward program:

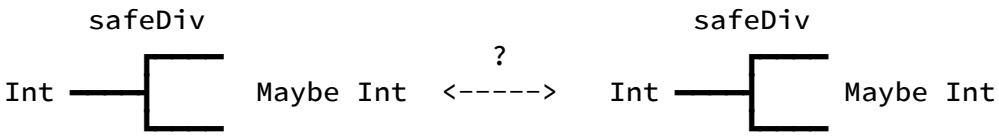
```
x, y, z :: Int
x = 123
y = (`div` 4) x
z = (`div` 5) y
```

However, we know that `div` is unsafe since dividing it by 0 gives a zero division error. Therefore, we should write a safe `div` function that returns `Nothing` if division by 0 is to be expected:

```
safeDiv x y :: Int -> Maybe Int
safeDiv x 0 = Nothing
safeDiv x y = div x y
```

However, composing `safeDiv` is now no longer straightforward:

```
x = 123
y = (`safeDiv` 4) x
z = ???
```



Let us try using `fmap`:

```
x :: Int
x = 123

y :: Maybe Int
y = (`safeDiv` 4) x

z :: Maybe (Maybe Int)
z = fmap (`safeDiv` 5) y
```

Although this typechecks, the resulting type `Maybe (Maybe Int)` is incredibly awkward. It tells us that there is potentially a `Maybe Int` term, which means that there is *potentially* a *potential* `Int`. What would be better is to collapse the `Maybe (Maybe Int)` into just `Maybe Int`.

For this, we introduce the notion of a *Monad*, which again, can be described by a typeclass with some rules governing their methods. The primary feature of a `Monad m` is that it is an `Applicative` where we can collapse an `m (m a)` into an `m a` in the most obvious way. However, for convenience's sake, Haskell defines the `Monad` typeclass in a slightly different (but otherwise equivalent) formulation[1]:

```
class Applicative m => Monad m where
    return :: a -> m a -- same as pure
    (>>=) :: m a -> (a -> m b) -> m b -- composition in context
```

These methods are governed by the following laws:

- Left identity: `return a >>= h = h a`
- Right identity: `m >>= return = m`
- Associativity: `(m >>= g) >>= h = m >>= (\x -> g x >>= h)`

`return` is practically the same as `pure` (in fact it is almost always defined as `return = pure`). Although the word `return` feels incredibly odd, we shall see very shortly why it was named this way. `>>=` is known as the *monadic bind*[1] [2], and allows us to perform computation in context on a term in context, thereby achieving *composition in context*.

`>>=` is somewhat similar to `fmap`, in that while `fmap` allows us to apply an `a -> b` onto an `f a`, `>>=` allows us to apply an `a -> m b` onto an `m a`.

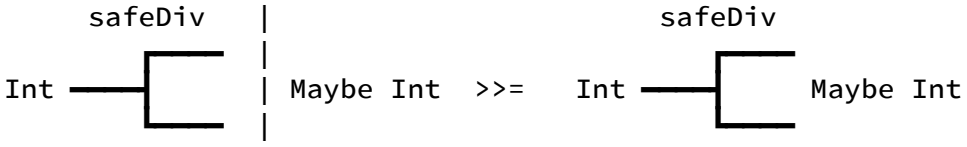
Let us see an instance of `Monad`:

```
instance Monad Maybe where
    return :: a -> Maybe a
    return = pure

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    Just x >>= f  = f x
```

With this instance, instead of using `fmap` to bring our `Maybe Int` into a `Maybe (Maybe Int)`, we can use `>>=` to just bring it to a `Maybe Int`!

```haskell
x :: Int
x = 123

y :: Maybe Int
y = (`safeDiv` 4) x

z :: Maybe Int
z = y >>= (`safeDiv` 5)
```

As we know, function composition `(g . f) x` is sort of to say "do `f` *and then* do `g` on `x`". Similarly, when `f` and `g` are computations in context and `x` is a term in context, `x >>= f >>= g` also means "do `f` *and then* do `g` on `x`"! However, `>>=` is incredibly powerful because the actual definition of `>>=` depends on the monad you use—therefore, monads allow us to *overload* composition in context![3]



Therefore, if you had `f :: a -> b` and `g :: b -> c` and `x :: a`, you would write `g (f x)` for `f` *and then* `g`. However, if you had `f :: a -> m b` and `g :: b -> m c` and `x :: m a`, you would write `x >>= f >>= g` for `f` *and then* `g`.

Beyond the Railways

As we know, data structures like `Maybe`, `Either` and `Validation` support the railway pattern, and them being functors, applicatives and (in the case of `Maybe` and `Either`) monads makes them ergonomic to use. However, the use of functors, applicatives and monads extend beyond just the railway pattern.

As described in Chapter 4.1 (Context/Notions of Computation), types like `[]` and `IO` provide *context* around a type. As it turns out, these types are also functors, applicatives and monads. While we have not touched `IO` at all so far, and will only do so in the next chapter, let us see the instance definitions for `[]`:

```
instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b]
    fmap = map

instance Applicative [] where
    pure :: a -> [a]
    pure x    = [x]

    (<*>) :: [a -> b] -> [a] -> [b]
    fs <*> xs = [f x | f <- fs, x <- xs]

instance Monad []  where
    return :: a -> [a]
    return = pure

    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]
```

Observe the definition of `>>=` for lists. The idea is that whatever `fmap f xs` produces (which is a 2+D list), `xs >>= f` flattens that result (it doesn't flatten it recursively, just the top layer). It does so by applying `f` onto every single `x` s in the list. As per the type signature, each `f x` produces a term of the type `[b]`, which is a list. We extract each `y` from that list, and put them all as elements of the resulting list. Let us see the action of `>>=` through an example:

```
ghci> fmap (\x -> return x) [1, 2, 3]
[[1], [2], [3]] -- fmap gives a 2D list
ghci> [1, 2, 3] >>= (\x -> return x)
[1, 2, 3]       -- >>= gives a 1D list

ghci> fmap (\x -> return (x, x + 1)) [1, 2, 3]
[[(1, 2)], [(2, 3)], [(3, 4)]] -- fmap gives a 2D list
ghci> [1, 2, 3] >>= (\x -> return (x, x + 1))
ghci> [(1, 2), (2, 3), (3, 4)] -- >>= gives a 1D list

ghci> [1, 2] >>= (\x -> [3] >>= (\y -> >>= return (x, y)))
[(1, 3), (2, 3)]
```

The last function can be written a little more clearly. Suppose we want to write a function that produces the "cartesian product" of two lists. Writing this function using the monad methods can look unwieldy, but will ultimately pay off as you will see shortly:

```
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys = xs >>= (\x ->
                          ys >>= (\y ->
                          return (x, y)))
```

As we expect, everything works!

```
ghci> cartesian_product [1,2] [3]
[(1,3),(2,3)]
```

Do-notation

The definition of `cartesian_product` above is hard to read. However, this form of programming is (as you will surely see) very common—we bind each `x` from `xs`, then bind each `y` from `ys`, and return `(x, y)`. Why not let us write the same implementation in this way:

```haskell
cartesian_product :: [a] -> [b] -> [(a, b)]
cartesian_product xs ys = do
    x <- xs
    y <- ys
    return (x, y)
```

Wouldn't this be much more straightforward? In fact, Haskell supports this! This is known as `do` notation, and is supported as long as the expression's type is a monad. `do` notation is just syntactic sugar for a series of `>>=` and lambda expressions:

```haskell
do e1 <- e2           ==>      e2 >>= (\e1 -> whatever code)
   whatever code
```

Therefore, the definition of `cartesian_product` using `do` notation is translated as follows:

```haskell
do x <- xs                 xs >>= (\x ->              xs >>= (\x ->
   y <- ys           ==>       do y <- ys        ==>   ys >>= (\y ->
   return (x, y)                   return (x, y))       return (x, y)))
```

More importantly, go back to the definition of `cartesian_product` using `do` notation. Compare that definition with the (more-or-less) equivalent definition in Python:

```python
def cartesian_product(xs, ys):
    for x in xs:
        for y in ys:
            yield (x, y)
```

What we have done was to **recover imperative programming with do-notation**! Even better: while `for` loops in Python only work on iterables, `do` notation in Haskell works on **any monad**!

```haskell
-- do notation with lists
pairs :: [a] -> [(a, a)]
pairs ls = do x <- ls
              y <- ls
              return (x, y)

-- do notation with Maybe
z :: Maybe Int
z = do y <- 123 `safeDiv` 4
       y `safeDiv` 5

-- do notation with Either
parseUser :: String -> String -> String -> Either String User
parseUser name email salary
  = do e <- parseEmail email
       s <- parseSalary salary
       return $ User name e s
```

Other languages like Python, C etc. define keywords like `for`, `while`, `if-else` as part of the language so that programmers can use different meanings of what *and then* means. For example, a `while` loop lets you write programs like (1) check condition, *and then* (2) if its true do the loop body, *and then* (3) check the condition again, etc. In Functional Programming languages like Haskell, it is *monads* that decide what *and then* means—this is great because **you** get to define your own monads and decide what composition of computation means!

```haskell
cartesian_product :: Monad m => m a -> m b -> m (a, b)
cartesian_product xs ys = do
    x <- xs
    y <- ys
    return (x, y)


ghci> cartesian_product [1, 2] [3]
[(1, 3), (2, 3)]
ghci> cartesian_product (Just 1) (Just 2)
Just (1, 2)
ghci> cartesian_product (Just 1) Nothing
Nothing
ghci> cartesian_product (Right 1) (Right 2)
Right (1, 2)
ghci> cartesian_product getLine getLine -- getLine is like input() in Python
alice -- user input
bob   -- user input
("alice","bob")
```

As you can tell, each monad has its own way of composing computation in context and has its own meaning behind the context it provides. This is why monads are such a powerful tool for functional programming! It is for this reason that we will dedicate the entirety of the next chapter to monads.

---

[1] You might notice that the monadic bind operator `>>=` looks very similar to the Haskell logo. Monads are incredibly important in functional programming, and we shall spend an entire chapter dedicated to this subject.

[2] Many popular languages call this `flatMap`.

[3] Just like how languages like C, C++ and Java have `;` to separate statements, i.e. a program like `A;B` means do `A` and then do `B`, `>>=` allows us to *overload* what *and then* means!

LAST UPDATED **26 OCT 2024**

Key Takeaways

- Instead of functions with side-effects, pure functions can emulate the desired effects (like branching railways) using the right data structures as notions of computation
- We can operate in context using regular functions when the context is a functor
- We can combine context when the context is an applicative
- We can compose functions in context sequentially when they are monads

Railway Pattern in Python

Aside from `do`-notation and all the niceties of programming with typeclasses, nothing else we have discussed in this chapter is exclusive to Haskell. In fact, many other languages have similar data structures to the ones we have seen, and are all functors and monads too! For example, we can implement `safeDiv` in `Java` using the built-in `Optional` class, which is the same as `Maybe` in Haskell, and to use its `flatMap` method instead of `>>=` in Haskell:

```
import java.util.Optional;
public class Main {
  static Optional<Integer> safeDiv(int num, int den) {
    if (den == 0) {
      return Optional.empty();
    }
    return Optional.of(num / den);
  }

  public static void main(String[] args) {
    Optional<Integer> x = safeDiv(123, 4)
        .flatMap(y -> safeDiv(y, 5))
        .flatMap(z -> safeDiv(z, 2));
    x.ifPresent(System.out::println);
  }
}
```

Therefore, what is required for using the railway pattern are

- the right data structures that have happy/sad paths, just like `Maybe`, `Either` and `Validation` (or even `[]`)
- the right methods so that they are functors, applicatives, monads etc, ensuring that they adhere to the laws as derived from category theory
- idiomatic *uses* of these data structures write pure functions, and to *use* their methods to concisely express functorial, applicative or monadic actions

Give these a try in the exercises!

LAST UPDATED 26 OCT 2024

Exercises

These exercises have questions that will require you to write code in Python and Haskell. All your Python code should be written in a purely-functional style.

Question 1

Create the following ADTs in Python:

- A singly linked list
- A `Maybe`-like type, with "constructors" `Just` and `Nothing`
- An `Either`-like type, with "constructors" `Left` and `Right`
- A `Validation`-like type, with "constructors" `Success` and `Failure`. Because Python does not have higher-kinds, you may assume that `Failure`s always hold a list of strings.

Then define methods on all these types so that they are all functors, applicatives and monads (`Validation` does not need to be a monad). `fmap` can be called `map`, `<*>` can be called `ap`, `return` can just be `pure`, and `>>=` can be called `flatMap`.

Due to Python's inexpressive type system, you are free to omit type annotations.

Try not to look at Haskell's definitions when doing this exercise to truly understand how these data structures work!

Example runs for each data structure follow:

Lists

```
# lists
>>> my_list = Node(1, Node(2, Empty()))

# map
>>> my_list.map(lambda x: x + 1)
Node(2, Node(3, Empty()))

# pure
>>> List.pure(1)
Node(1, Empty())

# ap
>>> Node(lambda x: x + 1, Empty()).ap(my_list)
Node(2, Node(3, Empty()))

# flatMap
>>> my_list.flatMap(lambda x: Node(x, Node(x + 1, Empty())))
Node(1, Node(2, Node(2, Node(3, Empty()))))
```

Maybe

```
>>> my_just = Just(1)
>>> my_nothing = Nothing()

# map
>>> my_just.map(lambda x: x + 1)
Just(2)
>>> my_nothing.map(lambda x: x + 1)
Nothing()

# pure
>>> Maybe.pure(1)
Just(1)

# ap
>>> Just(lambda x: x + 1).ap(my_just)
Just(2)
>>> Just(lambda x: x + 1).ap(my_nothing)
Nothing()
>>> Nothing().ap(my_just)
Nothing()
>>> Nothing().ap(my_nothing)
Nothing()

# flatMap
>>> my_just.flatMap(lambda x: Just(x + 1))
Just(2)
>>> my_nothing.flatMap(lambda x: Just (x + 1))
Nothing()
```

Either

```
>>> my_left = Left('boohoo')
>>> my_right = Right(1)

# map
>>> my_left.map(lambda x: x + 1)
Left('boohoo')
>>> my_right.map(lambda x: x + 1)
Right(2)

# pure
>>> Either.pure(1)
Right(1)

# ap
>>> Left('sad').ap(my_right)
Left('sad')
>>> Left('sad').ap(my_left)
Left('sad')
>>> Right(lambda x: x + 1).ap(my_right)
Right(2)
>>> Right(lambda x: x + 1).ap(my_left)
Left('boohoo')

# flatMap
>>> my_right.flatMap(lambda x: Right(x + 1))
Right(2)
>>> my_left.flatMap(lambda x: Right(x + 1))
Left('boohoo')
```

Validation

```
>>> my_success = Success(1)
>>> my_failure = Failure(['boohoo'])

# map
>>> my_failure.map(lambda x: x + 1)
Failure(['boohoo'])
>>> my_success.map(lambda x: x + 1)
Right(2)

# pure
>>> Validation.pure(1)
Right(1)

# ap
>>> Failure(['sad']).ap(my_success)
Failure(['sad'])
>>> Failure(['sad']).ap(my_failure)
Failure(['sad', 'boohoo'])
>>> Success(lambda x: x + 1).ap(my_success)
Success(2)
>>> Success(lambda x: x + 1).ap(my_failure)
Failure(['boohoo'])
```

Question 2

Question 2.1: Unsafe Sum

Recall Question 6 in Chapter 1.4 (Exercises) where we defined a function `sumDigits` in Haskell. Now write a function `sum_digits(n)` that does the same, i.e. sums the digits of a nonnegative integer n, in Python. Example runs follow:

```
>>> sum_digits(1234)
10
>>> sum_digits(99999)
45
```

Your Haskell definition should also run similarly:

```
ghci> sumDigits 1234
10
ghci> sumDigits 99999
45
```

Question 2.2: Safe Sum

Try entering negative integers as arguments to your functions. My guess is that something bad happens.

Let us make `sum_digits` safe. Re-define `sum_digits` so that we can drop the assumption that n is nonnegative (but will still be an integer), correspondingly using the `Maybe` context to keep our function pure. Use the `Maybe` data structure that you have defined from earlier for the Python version, and use Haskell's built-in `Maybe` to do so. Example runs follow:

```
>>> sum_digits(1234)
Just(10)
>>> sum_digits(99999)
Just(45)
>>> sum_digits(-1)
Nothing
```

```
ghci> sumDigits 1234
Just 10
ghci> sumDigits 99999
Just 45
ghci> sumDigits (-1)
Nothing
```

Question 2.3: Final Sum

Now define a function `final_sum(n)` that repeatedly calls `sum_digit` until a single-digit number arises. Just like your safe implementation of `sum_digit`, `final_sum` should also be safe. Example runs follow:

```
>>> final_sum(1234)
Just(1)
>>> final_sum(99999)
Just(9)
>>> final_sum(-1)
Nothing()
```

```
ghci> finalSum 1234
Just 1
ghci> finalSum 99999
Just 9
ghci> finalSum (-1)
Nothing
```

Tip: Use `do` -notation in your Haskell implementation!

Question 3

Question 3.1: Splitting Strings

Define a function `split` that splits a string delimited by a character. This is very similar to `s.split(c)` in Python. However, the returned result should be a singly-linked list—in Python, this would be the singly-linked-list implementation you defined in Question 1, and in Haskell, this would be just `[String]`.

Example runs follow:

```
>>> split('.', 'hello. world!. hah')
Node('hello', Node(' world!', Node(' hah', Empty())))
>>> split(' ', 'a   b')
Node('this', Node('', Node('', Node('is', Empty()))))
```

```
ghci> split '.' "hello. world!. hah"
["hello"," world!"," hah"]
ghci> split ' ' "a   b"
["a","","","b"]
```

Hint: The `split` function in Haskell was defined in the hands-on section in Chapter 4.4 (Railway Pattern#Validation).

Question 3.2: CSV Parsing

The Python `csv` library allows us to read CSV files to give us a list of rows, each row being a list of cells, and each cell is a string. Our goal is to do something similar using the list data structure.

A CSV-string is a string where each row is separated by `\n`, and in each row, each cell is separated by `,`. Our goal is to write a function `csv` that receives a CSV-string and puts all the cells in a two-dimensional list. Example runs follow.

```
>>> csv('a,b,c\nd,e\nf,g,h')
Node(Node('a', Node('b', Node('c', Empty()))),
Node(Node('d', Node('e', Empty()))),
Node(Node('f', Node('g', Node('h', Empty())))),
Empty()))
```

```
ghci> csv "a,b,c\nd,e\nf,g,h"
[["a","b","c"],["d","e"],["f","g","h"]]
```

Question 4

The formula $\binom{n}{k}$ is incredibly useful and has applications in domains like ~~gambling~~probability and statistics, combinatorics etc. The way to compute $\binom{n}{k}$ is straightforward:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Question 4.1: Factorial

Clearly, being able to compute factorials would make computing $\binom{n}{k}$ more convenient. Therefore, write a function `factorial` that computes the factorial of a nonnegative integer. Do so in Python and Haskell. Example runs follow.

```
>>> factorial(4)
24
>>> factorial(5)
120
```

```
ghci> factorial 4
24
ghci> factorial 5
120
```

Question 4.2: Safe Factorial

Just like we have done in Question 2, our goal is to make our functions safer! Re-define `factorial` so that we can drop the assumption that the integer is nonnegative. In addition, your function should receive the name of a variable so that more descriptive error messages can be emitted. Use the `Either` type. Again, do so in Python and Haskell. Example runs follow:

```
>>> factorial(4, 'n')
Right(24)
>>> factorial(5, 'k')
Right(120)
>>> factorial(-1, 'n')
Left('n cannot be negative!')
>>> factorial(-1, 'k')
Left('k cannot be negative!')
```

```
ghci> factorial 4 "n"
Right 24
ghci> factorial 5 "k"
Right 120
ghci> factorial (-1) "n"
Left "n cannot be negative!"
ghci> factorial (-1) "k"
Left "k cannot be negative!"
```

Question 4.3: Safe n choose k

Now let us use `factorial` to define $\binom{n}{k}$! Use the formula described at the beginning of the question and our `factorial` functions to define a function `choose` that receives integers n and k and returns $\binom{n}{k}$. Example runs follow:

```
>>> choose(5, 2)
Right(10)
>>> choose(-1, -3)
Left('n cannot be negative!')
>>> choose(1, -3)
Left('k cannot be negative!')
>>> choose(3, 6)
Left('n - k cannot be negative!')
```

```
ghci> choose 5 2
Right 10
ghci> choose (-1) (-3)
Left "n cannot be negative!"
ghci> choose 1 (-3)
Left "k cannot be negative!"
ghci> choose 3 6
Left "n - k cannot be negative!"
```

Question 4.4: n choose k With Validation

Notice that several things could go wrong with $\binom{n}{k}$! Instead of using `Either`, change the implementation of `factorial` so that it uses the `Validation` applicative instead. This is so that all the error messages are collected. Your `choose` function definition should not change, aside from its type. Example runs follow.

```
>>> choose(5, 2)
Success(10)
>>> choose(-1, -3)
Failure(['n cannot be negative!', 'k cannot be negative!'])
>>> choose(1, -3)
Failure(['k cannot be negative!'])
>>> choose(3, 6)
Failure(['n - k cannot be negative!'])
```

```
ghci> choose 5 2
Success 10
ghci> choose (-1) (-3)
Failure ["n cannot be negative!","k cannot be negative!"]
ghci> choose 1 (-3)
Failure ["k cannot be negative!"]
ghci> choose 3 6
Failure ["n - k cannot be negative!"]
```

Tip: With the `-XApplicativeDo` extension, you can actually use `do` notation on `Functor`s and `Applicative`s. Give it a try by defining `choose` using `do`-notation! For more information on the conditions for when you can use `Applicative` `do`-notation, see the GHC Users Guide.

Note: `Validation` is not included in Haskell's Prelude. You can use the `Validation` datatype definition and its supporting typeclass instances as defined in the hands-on portion of Chapter 4.4 (Railway Pattern#Validation).

LAST UPDATED **26 OCT 2024**

Monads are a frequently recurring construct in functional programming, declarative programming and computer science, especially in programming language and logical semantics. In this chapter, we dive deeper into programming with monads and some additional supported operations beyond `return` and `>>=` and how to use them. Additionally, we show some more frequently used monads that go beyond the railway pattern, and show how monads *themselves* can be composed using *monad transformers*.

LAST UPDATED 26 OCT 2024

More on Monads

Recall from Chapter 4.5 (Railway Pattern#Monads) that monads support *composition in context*. This idea extends beyond the composition of functions that each branch out to happy and sad paths in the railway pattern. As you have seen, other types like `[]` don't have much to do with the railway pattern, but is still a monad. This because as long as a type describes some *notion of computation*, it can be a monad which supports composition in context. We have also seen how this can be useful when the programming language supports easy monadic computations, for example, with Haskell's `do` notation.[1]

However, if you observe the definition of the `Monad` type class carefully (see GHC Base: Control.Monad), you might notice that there are more methods and monadic operations than just `return` and `>>=`.

Ignoring values

In an imperative programming language like Python, we can write standalone expressions as statements, primarily to perform some side-effects. For example:

```python
def my_function(x):
    print(x) # standalone statement
    return x
```

We can, in fact, write the `print` statement in the style of `z <- print x` in Haskell, although that would be useless since that variable's value is not used at all and is not meaningful to begin with:

```python
def my_function(x):
    z = print(x) # why?
    return x
```

Therefore, monads also have a method `>>` that basically discards the result of a monadic action. This method has the following type signature, which, in comparing with that of `>>=` should make this more apparent:

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>)  :: m a ->       m b  -> m b
```

As you can tell, unlike `>>=`, the second argument to `>>` is not a function, but is just another term of the monad. It ignores whatever `a` is in context in the first argument, and only uses it for sequencing with the second argument of type `m b`.

Thus, `do` notation actually uses `>>` when composing monadic operations when the result of an operation is to be discarded. We give some more rules of `do` notation, including the rules for translating `let` binds, which allows *pure* bindings, in contrast with `<-` which defines a monadic bind. Note that in `do` notation, there is no need to write `in` for `let` binds:

```
do s           ==>    s                          -- plain

do e1 <- e2    ==>    e2 >>= (\e1 -> do s)       -- monadic bind
   s

do e           ==>    e >> do s                  -- monadic bind, ignore
   s

do let x = e   ==>    let x = e in do s          -- pure bind
   s
```

For example, we have seen how `>>=` on lists performs a `for` loop of sorts. For lists, `>>` does more or less the same thing, except that the values in the previous list cannot be accessed. For example,

```
ghci> [1, 2] >>= (\x -> [(x, 3)])
[(1, 3), (2, 3)]
ghci> [1, 2] >>= (\_ -> [3])
[3, 3]
ghci> [1, 2] >> [3]
[3, 3]
```

Of course, `>>` on lists is not particularly useful, but we shall see some uses of `>>` for other monads shortly.

Monadic Equivalents of Functions

Due to the prevalence of monads, many of the familiar functions like `map` and `filter` have monadic equivalents. These are usually written with a postfix `M`, such as `mapM` or `filterM`. In addition, such functions can also ignore results and are written with a postfix `_`, such as `mapM_` or `filterM_`. We show what we mean by "monadic equivalent" by juxtaposing the type signatures of some familiar functions and their monadic counterparts:

```
map      ::            (a -> b)   -> [a] -> [b]
mapM @[] :: Monad m => (a -> m b) -> [a] -> m [b]

filter  ::            (a -> Bool)   -> [a] -> [a]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

Let us see some examples of `mapM` in action:

```
ghci> map (+2) [1, 2, 3]
[3, 4, 5]
ghci> map (Just . (+2)) [1, 2, 3]
[Just 3, Just 4, Just 5]
ghci> mapM (Just . (+2)) [1, 2, 3]
Just [3, 4, 5]
```

One example of `mapM` over lists and `Maybe` s is with validation. Let us suppose we want to read a list of strings as a list of integers. To start with, we can use a function `readMaybe` that attempts to parse a `String` into a desired data type:

```
ghci> import Text.Read
ghci> :{
ghci| toInt :: String -> Maybe Int
ghci| toInt = readMaybe
ghci| :}
ghci> toInt "123"
Just 123
ghci> toInt "hello"
Nothing
```

The `mapM` function allows us to ensure that all elements of a list of strings can be converted into `Int` s!

```
ghci> mapM toInt ["1", "2", "3"]
Just [1, 2, 3]
ghci> mapM toInt ["hello", "1", "2"]
Nothing
```

Monadic Controls

Another useful tool that comes with monads are control functions. For example, in an imperative program we might write something like the following:

```
def f(x):
    if x > 10:
        print(x)
    return x
```

In Haskell, since `if - else` statements are actually expressions and must have an `else` branch, we might have to write something like the following:

```
f x = do
    if x > 10
    then someAction x
    else return () -- basically does nothing
    return x
```

Notice the `return ()` expression. Because every "statement" in a `do` block must be monadic, we must write a monadic expression in every branch. In addition, we are clearly using `someAction` for its monadic effects, so the "returned" value is completely useless, likely just `()` (the unit type, which means nothing significant). Therefore, the corresponding `else` branch must also evaluate to `m ()` for whatever monad `m` we are working with. This is a chore and much less readable!

Instead, we can use regular functions to simulate `if ... then ...` statements in a monadic expression. This is the `when` function defined in `Control.Monad` [2]:

```
when :: Applicative f => Bool -> f () -> f ()
```

As you can tell, `when` receives a boolean condition and one monadic action and gives you a monadic action. Importantly, the monad wraps around `()`, which means that this operation is useful for some monadic effect, such as `IO`. This allows our function above to be written as:

```
import Control.Monad
f x = do
    when (x > 10) (someAction x)
    return x
```

Although later we will see that the monadic action `someAction` can actually cause side effects, it is not necessarily the case that side effects are the only reason why a monadic action `m ()` is useful. Another example of this is the `guard` function:

```
guard :: Alternative f :: Bool -> f ()
```

If the monad you are working with is also an `Alternative`, the `guard` function, essentially, places a guard (like guards in imperative programming) based on a condition, returning the sad path immediately if the condition fails. To see this in action, let us see how we can use `guard` to implement `safeDiv`:

```haskell
import Control.Monad

safeDiv1 :: Int -> Int -> Maybe Int
safeDiv1 x y = if y == 0
               then Nothing
               else Just (x `div` y)

safeDiv2 :: Int -> Int -> Maybe Int
safeDiv2 x y
    = do guard (y /= 0)
         return $ x `div` y
```

An `Alternative` is an applicative structure that has an `empty` case. For example, an `empty` list is `[]`, and an `empty Maybe` is `Nothing`. The definition of `guard` makes this really simple:

```haskell
guard :: Alterative f => Bool -> f ()
guard True = pure ()
guard False = empty
```

Notice how `guard` works in `safeDiv2`. If `y` is not `0`, then `guard (y /= 0)` evaluates to `Just ()`. Sequencing `Just ()` with `return $ x `div` y` gives `Just (x `div` y)`. However, if `y` is equal to `0`, then `guard (y /= 0)` evaluates to `Nothing`. We know that `Nothing >>= f` for any `f` will always give `Nothing`, so `Nothing >> x` will also always give `Nothing`. Therefore, `Nothing >> return (x `div` y)` will give us `Nothing`. As you can see, `guard` makes monadic control easy!

As before, `guard` works on any `Alternative`. For this reason, let us see how `guard` works in the `[]` monad:

```haskell
ghci> import Control.Monad
ghci> ls = [-2, -1, 0, 1, 2]
ghci> :{
ghci> ls2 = do x <- ls
ghci|          guard (x > 0)
ghci|          return x
ghci| :}
ghci> ls2
[1, 2]
```

As you can see, `guard` essentially places a filter on the elements of the list! This is because `[()] >> ls` just gives `ls`, whatever `ls` is, and `[] >> ls` just gives `[]`. In fact, `>>` over lists somewhat like the following function using a `for` loop in Python:

```
>>> def myfunction(ls2, ls):
...     x = []
...     for _ in ls2:
...         x.extend(ls)
...     return x
>>> my_function([()], [1, 2, 3])
[1, 2, 3]
>>> my_function([], [1, 2, 3])
[]
```

As you can tell, if `f` is `False`, then `guard f >> ls` will give `[]`; otherwise, it will just give `ls` itself. This makes it such that we now have a way to filter elements of a list! Better still, if we combined this with something else:

```
ghci> import Control.Monad
ghci> ls = [-2, -1, 0, 1, 2]
ghci> :{
ghci> ls2 = do x <- ls
ghci|          guard (x > 0)
ghci|          return $ x * 2
ghci| :}
ghci> ls2
[2, 4]
```

Notice how we have just recovered list comprehension! The definition of `ls2` can also be written as the following:

```
ghci> ls = [-2, -1, 0, 1, 2]
ghci> ls2 = [x * 2 | x <- ls, x > 0]
ghci> ls2
[2, 4]
```

Thus, as you can see, list comprehensions are just monadic binds and guards specialized to lists! Even better, `do` notation allows you to use `guards`, monadic binds etc. in any order and over any monad, giving you maximum control over how you write monadic programs.

---

[1] Other languages like Scala also have similar facilities for writing monadic computations. In fact, the Lean 4 programming language takes Haskell's `do` notation much further (Ullrich and de Moura; 2022).

[2] The monadic control functions described in this section are defined in the `Control.Monad` module in Haskell's `base` library, i.e., they need to be imported, but do not need to be installed (just like the `math` library in Python).

References

Sebastian Ullrich and Leonardo de Moura. 2022. `do` Unchained: Embracing Local Imperativity in a Purely Functional Language (Functional Pearl). *Proceedings of the ACM on Programming Languages (PACMPL)*. 6(ICFP) Article 109 (August

2022), 28 pages. URL: https://doi.org/10.1145/3547640.

LAST UPDATED **26 OCT 2024**

Commonly Used Monads

Thus far, we have looked at monads like `[]`, `Maybe` and `Either`. Recall that these monads describe the following notions of computation:

- `[]`: nondeterminism
- `Maybe`: potentially empty computation
- `Either a`: potentially failing computation

However, there are many more monads that you will frequently encounter, and in fact, many libraries (even in other programming languages) expose classes or data types that work as monads. Most of these monads involve one or both of the following notions of computation:

1. Reading from state
2. Writing to, or editing state

In fact, side effects can also be seen as reading from and writing to state. In this section, we shall describe some commonly used monads that implement these ideas.

Reader

A very common pattern of computation is *reading from state*, i.e. performing computations based on some environment. For example, we may have a local store of users in an application, from which we retrieve some user information and do stuff with it. Typically, this is represented by a plain function of type `env -> a`, where `env` is the environment to read from, and `a` is the type of the result that depends on the environment. For example, we can determine if two nodes are connected in a graph by using depth-first search—however, connectivity of two nodes depends on the graph, where two nodes might be connected in one graph, but not in another. Therefore, the result of a depth-first search depends on the graph. However, depth-first search requires us to look up the neighbours of a node so that we can recursively search them, thereby also depending on the graph. As such, we want some way to compose two functions that receive a graph (monadically).

In general, we can let any term of type `env -> a` be seen as a term of type `a` that depends on an environment `env`. In other words, the type `env -> ?` describes the notion of computation of something depending on an environment. And as it turns out, for any environment type `env`, the partially applied type `(->) env` i.e. `env -> a` for all `a` is a `Monad`!

```haskell
instance Functor ((->) env) where
    fmap :: (a -> b) -> (env -> a) -> (env -> b)
    fmap f x = f . x

instance Applicative ((->) env) where
    pure :: a -> (env -> a)
    pure = const
    (<*>) :: (env -> (a -> b)) -> (env -> a) -> env -> b
    (<*>) f g x = f x (g x)

instance Monad ((->) env) where
    return :: a -> (env -> a)
    return = pure
    (>>=) :: (env -> a) -> (a -> (env -> b)) -> env -> b
    (>>=) m f x = f (m x) x
```

The definition of `fmap` is incredibly straightforward, essentially just doing plain function composition. The definition of `pure` is just `const`, where `const` is defined to be `const x = \_ -> x`, i.e. `pure` receives some value and produces a function that ignores the environment and produces that value. `<*>` takes two functions `f` and `g` and performs applicative application by applying each of them to the same environment `x`. Most notably, `<*>` applies the same environment *unmodified* to both functions. Finally, `>>=` operates pretty similar to `<*>` except with some changes to how the functions are applied.

For clarity, let's define a type alias `Reader env a` which means that it is a type that reads an environment of type `env` and returns a result of type `a`:

```haskell
type Reader = (->)
```

Then, let's try to implement depth-first search with the `Reader` monad. First, we define some additional types, like the graph, which for us, has nodes as integers, and is represented using an adjacency list:

```haskell
type Node = Int
type Graph = [(Node, [Node])]
```

Next, we define a function `getNeighbours` which gets the nodes that are adjacent to a node in the graph:

```haskell
getNeighbours :: Node -> Reader Graph [Node]
getNeighbours x = do
    neighbours <- lookup x
    return $ concat neighbours
```

Notice that our `getNeighbours` function does not refer to the graph at all! We can just use `do` notation, and Haskell knows how to compose these computations!

Using `getNeighbours`, we can now define `dfs` which performs a depth-first search via recursion:

```
dfs :: Node -> Node -> Reader Graph Bool
dfs src dst = aux [] src where
  aux :: [Node] -> Node -> Reader Graph Bool
  aux visited current
    | arrived         = return True
    | alreadyVisited  = return False
    | otherwise       = do
      neighbours <- getNeighbours current
      ls <- mapM (aux (current : visited)) neighbours
      return $ or ls
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

Let us learn how this works. Within the `dfs` function we define an auxiliary function that has a `visited` parameter. This is so that a user using the `dfs` function will not have to pass in the empty list as our `visited` "set". The `aux` function is where the main logic of the function is written. The first two cases are straightforward: (1) if we have arrived at the destination then we return `True`, and (2) if we have already visited the current node then we return `False`. If both (1) and (2) are not met, then we must continue searching the graph. We first get the neighbours of the current node using the `getNeighbours` function, giving us `neighbours`, which are the neighbours of the current node. Then, we recursively map `aux` (thereby recursively performing `dfs`) over all the neighbours. However, since `aux` is a monadic operation, we use `mapM` to map over the neighbours, giving us a list of results. We finally just check whether any of the nodes give us a positive result using the `or` function, corresponding to the `any` function in Python. Note one again that our `dfs` function makes no mention of the map at all, and we do not even need to pass the map into `getNeighbours`! The `Reader` monad automatically passes the same environment into all the other `Reader` terms that receive the environment.

Using the `dfs` function is very simple. Since the `Reader` monad is actually just a function that receives an environment and produces output, to use a `Reader` term, we can just pass in the environment we want!

```
ghci> my_map = [(1, [2, 3])
               , (2, [1])
               , (3, [1, 4])
               , (4, [3])
               , (5, [6])
               , (6, [5])]
ghci> dfs 5 6 my_map
True
ghci> dfs 5 2 my_map
False
ghci> dfs 1 2 [] -- empty map
False
```

Finally, note that we can retrieve the environment directly within the `Reader` monad by just using the identity function `id`!

```
ask :: Reader env env
ask = id

getNeighbours :: Node -> Reader Graph [Node]
getNeighbours x = do
    my_graph <- ask -- gets the graph directly
    let neighbours = lookup x my_graph
    return $ concat neighbours
```

Writer

The dual of a `Reader` is a `Writer`. In other words, instead of reading from some state or environment, the `Writer` monad has state that it writes to. The simplest example of this is logging. When writing an application, some (perhaps most) operations should be logged, so that we developers have usage information, crash dumps and so on, which can be later analysed.

In general, we can let any term of type `(log, a)` be seen as a type `a` that also has a log `log`. And as it turns out, for any log type `log`, the partially applied type `(log,)`, i.e. `(log, a)` for all `a` is a `Monad`!

```
instance Functor (log,) where
    fmap :: (a -> b) -> (log, a) -> (log, b)
    fmap f (log, a) = (log, f a)

instance Monoid log => Applicative (log,) where
    pure :: a -> (log, a)
    pure = (mempty,)
    (<*>) :: (log, a -> b) -> (log, a) -> (log, b)
    (<*>) (log1, f) (log2, x) = (log1 `mappend` log2, f x)

instance Monad (log,) where
    return :: a -> (log, a)
    return = pure
    (>>=) :: (log, a) -> (a -> (log, b)) -> (log, b)
    (log, a) >>= f = let (log2, b) = f a
                     in  (log1 `mappend` log2, b)
```

Let's carefully observe what the instances say. The `Functor` instance is straightforward—it applies the mapping function onto the second element of the tuple. The `Applicative` and `Monad` instances are more interesting. Importantly, just like the definition of the `Applicative` instance for `Validation`, the two logs are to be combined via an associative binary operation `<>`, which in this case is `mappend`. In most occasions, `mappend` is the same as `<>`. However, applicatives must also have a `pure` operation. In the case of `Either` and

`Validation`, `pure` just gives a `Right` or `Success`, therefore not requiring any `log`. However, in a tuple, we need some "empty" `log` to add to the element to wrap in the tuple.

Thus, the `log` not only must have an associative binary operation, it needs some "empty" term that acts as the identity of the binary operation:

$$E \oplus \text{empty} = \text{empty} \oplus E = E$$

$$E_1 \oplus (E_2 \oplus E_3) = (E_1 \oplus E_2) \oplus E_3$$

This is known as a `Monoid`, which is an extension of `Semigroup`!

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

Typically, `mappend` is defined as `<>`.

Recall that `[a]` with concatenation is a `Semigroup`. In fact, `[a]` is also a `Monoid`, where `mempty` is the empty list!

```
ls ++ [] = [] ++ ls = ls
x ++ (y ++ z) = (x ++ y) ++ z
```

Therefore, as long as `a` is a `Monoid`, then `(a, b)` is a monad!

Lastly, just like how `Reader`s have an `ask` function which obtains the environment, `Writer`s have a `write` function which writes a message to your log—the definition of `write` makes this self-explanatory.

```
write :: w -> (w, ())
write = (,())
```

Let us see this monad in action. Just like with `Validation`, we are going to let `[String]` be our log.

```
type Writer = (,)
type Log = [String]
```

Then, we write an example simple function that adds a log message:

```
loggedAdd :: Int -> Int -> Writer Log Int
loggedAdd x y = do
    let z = x + y
    write [show x ++ " + " ++ show y ++ " = " ++ show z]
    return z
```

Composing these functions is, once again, incredibly straightforward with `do` notation!

```haskell
loggedSum :: [Int] -> Writer Log Int
loggedSum [] = return 0
loggedSum (x:xs) = do
    sum' <- loggedSum xs
    loggedAdd x sum'
```

With this, the `loggedSum` function receives a list of integers and returns a pair containing the steps it took to arrive at the sum, and the sum itself:

```haskell
ghci> y = loggedSum [1, 2, 3]
ghci> snd y
6
ghci> fst y
["3 + 0 = 3","2 + 3 = 5","1 + 5 = 6"]
```

State

However, many times, we will also want to compose functions that do *both* reading from, and writing to or modifying state. In essence, it is somewhat a combination of the `Reader` and `Writer` monads we have seen. One example is pseudorandom number generation. A pseudorandom number generator receives a seed, and produces a random number and the next seed, which can then be used to generate more random numbers. The type signature of a pseudorandom number generation function would be something of the form:

```haskell
randomInt :: Seed -> (Int, Seed)
```

This pattern extends far beyond random number generation, and can be used to encapsulate the idea of a stateful transformation. For this, let us define a type called `State`:

```haskell
newtype State s a = State { runState :: s -> (a, s) }
```

Notice the `newtype` declaration. A `newtype` declaration is basically a `data` declaration, except that it must have exactly one constructor with exactly one field. In other words, a `newtype` declaration is a wrapper over a single type, in our case, the type `s -> (a, s)`. `newtype`s only differ from their wrapped types while programming and during type checking, but have no operational differences—after compilation, `newtype`s are represented exactly as the type they wrap, thereby introducing no additional overhead. However, `newtype` declarations also behave like `data` declarations, which allow us to create a new type from the types they wrap, allowing us to give new behaviours to the new type.

With this in mind, let us define the `Monad` instance for our `State` monad:

```haskell
instance Functor (State s) where
    fmap :: (a -> b) -> State s a -> State s b
    fmap f (State f') = State $
        \s -> let (a, s') = f' s
              in  (f a, s')

instance Applicative (State s) where
    pure :: a -> State s a
    pure x = State (x,)
    (<*>) :: State s (a -> b) -> State s a -> State s b
    (<*>) (State f) (State x) = State $
        \s -> let (f', s') = f s
                  (x', s'') = x s'
              in  (f' x', s'')

instance Monad (State s) where
    return :: a -> State s a
    return = pure
    (>>=) :: State s a -> (a -> State s b) -> State s b
    (State f) >>= m = State $
        \s -> let (a, s') = f s
                  State f' = m a
              in  f' s'
```

The instance definitions are tedious to define. Furthermore, nothing worthy of note is defined—the methods implement straightforward function composition. However, it is these methods that allow us to compose stateful computation elegantly!

Finally, just like `ask` for `Reader`s and `write` for `Writer`s, we have `get` and `put` to retrieve and update the state of the monad accordingly, and an additional `modify` function which modifies the state:

```haskell
put :: s -> State s ()
put s = State $ const ((), s)

get :: State s s
get = State $ \s -> (s, s)

modify :: (s -> s) -> State s ()
modify f = do s <- get
              put (f s)
```

Let's try this with an example. Famously, computing the fibonacci numbers in a naive recursive manner is incredibly slow. Instead, by employing memoization, we can take the time complexity of said function from $\mathrm{O}(2^n)$ down to $\mathrm{O}(n)$. Memoization requires retrieving and updating state, making it an ideal candidate for using the `State` monad!

We first define our state to be a table storing inputs and outputs of the function. Then, writing the fibonacci function is straightforward. Note the use of `Integer` instead of `Int` so that we do not have integer overflow issues when computing large fibonacci numbers:

```haskell
type Memo = [(Integer, Integer)]
getMemoized :: Integer -> State Memo (Maybe Integer)
getMemoized n = lookup n <$> get

fib :: Integer -> Integer
fib n = fst $ runState (aux n) [] where
  aux :: Integer -> State Memo Integer
  aux 0 = return 0
  aux 1 = return 1
  aux n = do
    x <- getMemoized n
    case x of
        Just y -> return y
        Nothing -> do
            r1 <- aux (n - 1)
            r2 <- aux (n - 2)
            let r = r1 + r2
            modify ((n, r) :)
            return r
```

The `getMemoized` function essentially just performs a lookup of the memoized input from the state. Then, the `fib` function defines an auxiliary function `aux` like before, which contains the main logic describing the computation of the fibonacci numbers. In particular, the `aux` function returns `State Memo Integer`. As such, to access the underlying state processing function produced by `aux n`, we must use the `runState` accessor function as defined in the `newtype` declaration for `State`. `runState (aux n)` gives us a function `Memo -> (Integer, Memo)`, and thus passing in the empty memo (`runState (aux n) []`) gives us the result. The result is a pair `(Integer, Memo)`, and since we do not need the memo after the result has been computed, we just discard it and return it from `fib`.

The `aux` function is similarly straightforward, with the usual two base cases. In the recursive case `aux n`, we first attempt to retrieve any memoized result using the `getMemoized` function. If the result has already been computed (`Just y`), then we return the memoized result directly. Otherwise, we recursively compute `aux (n - 1)` and `aux (n - 2)`. Importantly, `aux (n - 1)` will perform updates to the state (the memo), which is then passed along automatically (via monadic bind) to the call to `aux (n - 2)`, eliminating the exponential time complexity. Once `r1` and `r2` have been computed, the final result is `r`. Of course, we add the entry `n -> r` into the memo, and we can do so using the `modify` function, where `modify ((n, r) :)` prepends the pair `(n, r)` onto the memo. Of course, we finally return `r` after all of the above has been completed.

The result of this is polynomial-time `fib` function that can comfortably compute large fibonacci numbers:

```
ghci> fib 1
1
ghci> fib 5
5
ghci> fib 10
55
ghci> fib 20
6765
ghci> fib 100
354224848179261915075
ghci> fib 200
280571172992510140037611932413038677189525
```

I/O

Until now, we still have no idea how Haskell performs simple side effects like reading user input or printing to the console. In fact, nothing we have discussed so far involves side effects, because Haskell is a purely functional programming language, and all functions are pure. One of the key innovations of monads is that it allows a purely functional programming language like Haskell to produce side effects... but how?

Typically, a function that produces side effects is a regular function, except that it will also cause some additional effects on the side. One example is the `print` function in Python, which has the following type signature:

```
def print(x: object) -> NoneType: # prints to the console
    # ...
```

However, notice that the `State` monad is somewhat similar. A term of `State s a` wraps a function `s -> (a, s)`; it is a pure function that is meant to compute a term of type `a`. However, it has the additional effect of depending on some state of type `s`, and will also produce some new state also of type `s`. Therefore, `State s a` can be seen as an impure function/term of type `a`, with the side effect of altering state.

What if the state `s` was actually the real world itself? In essence, the function `RealWorld -> (a, RealWorld)` is a function that receives the real world (as in, literally the world), and produces some term `a` and a new state of the world? In this view, a function that prints to the console receives the current state of the world and computes nothing (just like how `print` in Python returns `None`), and also produces the new state of the world where text has been printed to the console. Then, `input` in Python can be seen as a function that receives a state of the world containing user input, and produces the value entered by the user, retaining the current state of the world! These functions can thus actually be seen as *pure functions*, as long as we view the real world as a term in our programming language! In essence:

> The `IO` monad is the `State` monad where the state is the real world.

This is how Haskell, a purely functional programming language, performs I/O, a side effect. In fact, our characterization of `IO` is not merely an analogy, but is exactly how `IO` is represented in Haskell:

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

As such, after learning how the `State` monad works, performing I/O in Haskell should now be straightforward, especially with `do` notation. Let us finally, after five chapters, write a "Hello World" program.

```
main :: IO ()
main = putStrLn "Hello World!"
```

The `putStrLn` function has type `String -> IO ()`. It essentially receives a string to print, and alters the state of the world by adding the string to the console.

Importantly, every Haskell program can be seen as the `main` function, which has type `IO ()`. Recall that `IO` is just the `State` monad, which wraps a function that receives the state of the real world at function application, and produces a new state of the world and some other pure computation. In essence, the `main` function therefore has type `State# RealWorld -> (# State# RealWorld, () #)`. Therefore, we can see, roughly, that when a Haskell program is run, the current state of the world is passed into `main`, giving us a new state of the world where the program has completed execution!

Just like the `State` monad, we can compose `IO` operations monadically with `do` notation. For example, the `getLine` function has type `IO String`, similar to `input` in Python except it does not receive and print a prompt. Thus, we can write a program that reads the name of a user and says hello to that user like so:

```
-- Main.hs
main :: IO ()
main = do
    name <- getLine
    putStrLn $ "Hello " ++ name ++ "!"
```

Now, instead of loading the program with GHCi, we can *compile* this program with GHC into an executable!

```
ghc Main.hs
```

When we run the program, the program waits for us to enter a name, then says hello to us!

```
Yong Qi
Hello Yong Qi!
```

Other `IO` operations can be found in Haskell's Prelude, and these should be relatively straightforward to understand.

LAST UPDATED **26 OCT 2024**

Monad Transformers

Monads support *composition in context*. Another question to ask is, can we *compose monads*? In other words, can we combine monads together?

Consider the example of finding the length of the path between two connected neighbours in a directed graph, except that we have each node connected to at most one edge. The way we might solve this problem is, once again, via DFS (which in this case is the same as BFS), except that our graph is now of type `[(Node, Node)]` and our function returns the length of the path instead of a `Bool` value describing whether the path exists:

```
type Node = Int
type Graph = [(Node, Node)]
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst gph = aux src [] gph where
    aux :: Node -> [Node] -> Graph -> Maybe Int
    aux current visited gph'
      | arrived = return 0
      | alreadyVisited = Nothing
      | otherwise  = do
          n <- lookup current gph
          (+1) <$> aux n (current : visited) gph'
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

Notice that just like our previous definition of `dfs`, all our functions such as `dfs` and `lookup` involve some environment which we need to pass around! Let us try changing everything of type `Graph -> Maybe Int` to `Reader Graph (Maybe Int)` and modify our environment to no longer receive the `gph` argument:

```
type Node = Int
type Graph = [(Node, Node)]
dfs :: Node -> Node -> Reader Graph (Maybe Int)
dfs src dst = aux src [] where
    aux :: Node -> [Node] -> Reader Graph (Maybe Int)
    aux current visited
      | arrived = return 0
      | alreadyVisited = Nothing
      | otherwise  = do
          n <- lookup current
          (+1) <$> aux n (current : visited)
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

Unfortunately, our code doesn't type check. This is because now our `do` block performs the monadic operations based on the definition of `Reader`, not on `Maybe`! As such, we may

need significant rewrites to our function to introduce the `Reader` monad to our `Maybe` computation.

Enriching the `Maybe` Monad

Is there a better way? Yes! Let us try defining a new monad `ReaderMaybe` that essentially acts as both the `Reader` and the `Maybe` monads!

```
newtype ReaderMaybe env a = ReaderMaybe { runReaderMaybe :: Reader env (Maybe
a) }

instance Functor (ReaderMaybe env) where
  fmap :: (a -> b) -> ReaderMaybe env a -> ReaderMaybe env b
  fmap f (ReaderMaybe ls) = ReaderMaybe $ fmap (fmap f) ls

instance Applicative (ReaderMaybe env) where
  pure :: a -> ReaderMaybe env a
  pure = ReaderMaybe . pure . pure
  (<*>) :: ReaderMaybe env (a -> b) -> ReaderMaybe env a -> ReaderMaybe env b
  (ReaderMaybe f) <*> (ReaderMaybe x) = ReaderMaybe $ do
    maybe_f <- f
    case maybe_f of
      Nothing -> return Nothing
      Just f' -> do
        maybe_x <- x
        case maybe_x of
          Nothing -> return Nothing
          Just x' -> return $ Just (f' x')

instance Monad (ReaderMaybe env) where
  return :: a -> ReaderMaybe env a
  return = pure
  (>>=) :: ReaderMaybe env a -> (a -> ReaderMaybe env b) -> ReaderMaybe env b
  (ReaderMaybe ls) >>= f = ReaderMaybe $ do
    m <- ls
    case m of
      Just x -> runReaderMaybe $ f x
      Nothing -> return Nothing
```

All of these methods are tedious to define, however are somewhat straightforward. In particular, it relies on `do` notation on `Reader`s to extract out the `Maybe` values, and performs the usual `Maybe` methods to compose them.

The result is that we can now make use of this `ReaderMaybe` monad in our `dfs` function:

```haskell
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst = runReaderMaybe (aux src []) where
    aux :: Node -> [Node] -> ReaderMaybe Graph Int
    aux current visited
      | arrived = return 0
      | alreadyVisited = ReaderMaybe $ return Nothing
      | otherwise  = do
          n <- ReaderMaybe $ lookup current
          (+1) <$> aux n (current : visited)
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

There are several points worthy of note in our new implementation:

1. Most of this definition is the same as our original definition that works on the `Maybe` monad
2. Because the `aux` function returns a `ReaderMaybe` term which wraps the actual `Reader` function, we write `runReaderMaybe (aux src [])` to expose the actual `Reader Graph (Maybe Int)` function
3. In the `alreadyVisited` case, we cannot write `alreadyVisited = Nothing` since `Nothing` is not of the type `ReaderMaybe Graph Int`; we also cannot just write `return Nothing` since that has type `ReaderMaybe env (Maybe a)`. As such, we have to use `return @(Reader Graph) Nothing`, then wrap it in the `ReaderMaybe` constructor
4. Similar to (3), instead of `lookup current`, we have to wrap it around the `ReaderMaybe` constructor so that instead of having type `Reader Graph (Maybe Int)`, `ReaderMaybe $ lookup current` will have type `ReaderMaybe Graph Int`, which is the correct type to have.

When converting the original implementation based on `Maybe` into the new implementation based on `ReaderMaybe Graph Int`, one tip is to leave the implementation the same and just change the type signature of the functions to use `ReaderMaybe Graph Int` instead of `Graph -> Maybe Int`, then make use of typing information to correct the types in the program; in other words, "let the types guide your programming", like we have done in Chapter 2 (Types)! Furthermore, we are generally assured that everything works as expected because monads behave in the *most obvious way*!

Just like that, we are able to compose the `Reader` monad with the `Maybe` monad! Running `dfs` works exactly as we'd expect:

```haskell
ghci> my_map = [(1, 2), (2, 3), (3, 1)]
ghci> dfs 1 4 my_map
Nothing
ghci> dfs 1 2 my_map
Just 1
ghci> dfs 2 1 my_map
Just 2
```

Now, what if we wanted to enrich the `Maybe` monad with other notions of computation, such as `[]`, `IO` etc? Suppose we follow the same procedure of enriching `Maybe` with `Reader`, but instead by enriching it with `IO`, giving us a new monad `IOMaybe a` which represents `IO (Maybe a)`:

```
newtype IOMaybe a = IOMaybe { runIOMaybe :: IO (Maybe a) }

instance Functor IOMaybe where
  fmap :: (a -> b) -> IOMaybe a -> IOMaybe b
  fmap f (IOMaybe io) = IOMaybe (fmap (fmap f) io)

instance Applicative IOMaybe where
  pure :: a -> IOMaybe a
  pure = IOMaybe . pure . pure
  (<*>) :: IOMaybe (a -> b) -> IOMaybe a -> IOMaybe b
  (IOMaybe f) <*> (IOMaybe x) = IOMaybe $ do
    maybe_f <- f
    case maybe_f of
      Nothing -> return Nothing
      Just f' -> do
        maybe_x <- x
        case maybe_x of
          Nothing -> return Nothing
          Just x' -> return $ Just (f' x')

instance Monad IOMaybe where
  return :: a -> IOMaybe a
  return = pure
  (>>=) :: IOMaybe a -> (a -> IOMaybe b) -> IOMaybe b
  (IOMaybe m) >>= f = IOMaybe $ do
    maybe_m <- m
    case maybe_m of
      Just x -> runIOMaybe $ f x
      Nothing -> return Nothing
```

There are several things worth thinking about. Firstly, so far, it appears that we have to re-create new instances for *every* notion of computation we want to enrich `Maybe` with. Secondly, you might realise that absolutely nothing about the definition of the instances care about the enriching monad. All of the definitions in the methods for `ReaderMaybe` and `IOMaybe` do not mention any `Reader`-specific or `IO`-specific functions. Instead, they all rely on their respective monad binds! Therefore, we can abstract these into a *monad transformer*.

Monad Transformers

A monad transformer `MonadT m a` enriches `Monad` with `m`. For example, the `MaybeT m a` monad transformer enriches `Maybe` with `m`. Therefore, our `ReaderMaybe` and `IOMaybe` monads can be represented exactly as `MaybeT (Reader env)` and `MaybeT IO`! The definition of `MaybeT` is virtually the exact same as the definitions of `ReaderMaybe` and `IOMaybe`, except that we do not refer to `Reader` or `IO`, and leave them as `m`:

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Functor m) => Functor (MaybeT m) where
    fmap f (MaybeT x) = MaybeT $ fmap (fmap f) x

instance (Functor m, Monad m) => Applicative (MaybeT m) where
    pure = MaybeT . return . Just
    mf <*> mx = MaybeT $ do
        mb_f <- runMaybeT mf
        case mb_f of
            Nothing -> return Nothing
            Just f  -> do
                mb_x <- runMaybeT mx
                case mb_x of
                    Nothing -> return Nothing
                    Just x  -> return (Just (f x))

instance (Monad m) => Monad (MaybeT m) where
    return = MaybeT . return . Just
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)
```

With this `Maybe` monad transformer, we can rewrite our definition of `dfs` by replacing `ReaderMaybe Graph Int` with `MaybeT (Reader Graph) Int`!

```haskell
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst = runMaybeT (aux src []) where
    aux :: Node -> [Node] -> MaybeT (Reader Graph) Int
    aux current visited
      | arrived = return 0
      | alreadyVisited = MaybeT $ return Nothing
      | otherwise  = do
          n <- MaybeT $ lookup current
          (+1) <$> aux n (current : visited)
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

And now with the `MaybeT` monad transformer, we can enrich the `Maybe` monad with any other monad we want without having to redefine new types and new type class instances for each of the monads we are enriching `Maybe` with!

Monad Transformer Library

Because monads are so common in programming, the common monads already have their own monad transformers, and these are defined in the `transformers` and `mtl` libraries. If you want to use these commonly used monad transformers, just download the dependencies and `import` the libraries into your programs! But... how do we do that?

Build Tools and Package Managers

Most production programming languages have a package manager and build tool, and Haskell is no different. In fact, Haskell has *several* package managers and build tools you can use. Two of the main competing ones are `cabal` and `stack`, both of which can be installed via GHCup. For our purposes, we shall just use `cabal` since it is slightly simpler to use; most modern versions are generally fine, but for us, we shall use (at least) `cabal-3.10.3`.

Project Initialization

Using `cabal` is very simple. First, to create a new Haskell project, create an empty directory and run `cabal init` (`>` is the shell prompt of the terminal, do not enter `>` as part of the command)

```
> mkdir my-project
> cd my-project
> cabal init
```

Then, `cabal` will take you through a series of questions to initialize the project. Some notable options are:

- Executables are programs that can be executed; libraries are code that other Haskell users can import. For us, choose to build an executable
- The main module of the executable should be `Main.hs`. The `Main.lhs` option is for writing literate Haskell programs. You can use that as well, although for us, it is significantly easier to just use `Main.hs` and write plain Haskell programs.
- The language for our executable should be GHC2021, giving us as many of the latest features as we can have without having to include them as language extensions.

The result of running `cabal init` is that your project directory has been initialized with several parts:

- The `app` directory (or whatever name you have chosen) stores the source code of your program
- `my-project.cabal` is the specification of your project.

Project Configuration

Let us investigate what is in `my-project.cabal` (some comments and fields omitted for concision):

```
cabal-version:      3.0
-- ...
common warnings
    ghc-options: -Wall
executable my-project
    import:           warnings
    main-is:          Main.hs

    -- Modules included in this executable, other than Main.
    -- other-modules:

    -- LANGUAGE extensions used by modules in this package.
    -- other-extensions:

    -- Other library packages from which modules are imported.
    build-depends:    base ^>=4.17.2.1

    -- Directories containing source files.
    hs-source-dirs:   app

    -- Base language which the package is written in.
    default-language: GHC2021
```

The `executable my-project` clause describes some of the specifications of our project. In particular, the `build-depends` field describes any external dependencies we wish to include. These dependencies can be automatically pulled from Hackage by `cabal`, as long as we specify the name, and optionally the version, of the package. For example, we want the `Control.Monad.Trans.Maybe` module in `transformers` library. Hence, to include the `transformers` library to have access to monad transformers, just include `transformers` in `build-depends`.

```
-- ...
executable my-project
    -- ...
    -- Other library packages from which modules are imported.
    build-depends:    base ^>=4.17.2.1
                    , transformers
    -- ...
```

Then, run `cabal install` to install all our dependencies!

```
> cabal install
/path/to/my-project-0.1.0.0.tar.gz
Resolving dependencies...
Symlinking 'my-project' to '/path/to/.local/bin/my-project'
```

And that's all! Just like that, we now have access to `transformers` functions, data types, classes and methods!

Writing the Program

Let us try creating a simple executable program in our project. First, we create our simple graph library. Right now, our project directory looks like this:

```
my-project/
├─ my-project.cabal
├─ app/
│   └─ Main.hs
└─ ...
```

Let us create a simple graph library by creating a file `my-project/app/Data/Graph.hs`, therefore our directory structure becomes:

```
my-project/
├─ my-project.cabal
├─ app/
│   ├─ Main.hs
│   └─ Data/
│       └─ Graph.hs
└─ ...
```

This creates a new module called `Data.Graph`. We must include this in our `cabal` file so that `cabal` knows to compile it as well. Head back to `my-project.cabal`, and include `Data.Graph` in the `other-modules` field:

```
-- ...
executable my-project
    -- ...
    -- Modules included in this executable, other than Main.
    other-modules:    Data.Graph
    -- ...
```

Now, open `Graph.hs` and write some code! In particular:

1. Declare the name of the module. In this case, the module is called `Data.Graph` because it is in the `Data` directory and the file name is `Graph.hs`.
2. Import the `Control.Monad.Trans.Maybe` module to have access to `MaybeT`, and the `Control.Monad.Trans.Reader` monad to have access to the `Reader` monad.
3. Define our `dfs` function.

```haskell
module Data.Graph where

import Control.Monad.Trans.Maybe
import Control.Monad.Trans.Reader

type Graph = [(Node, Node)]
type Node = Int

type GraphProcessor = MaybeT (Reader Graph) Int

dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst = runReader $ runMaybeT (aux src []) where
    aux :: Node -> [Node] -> GraphProcessor
    aux current visited
      | arrived = return 0
      | alreadyVisited = MaybeT $ return Nothing
      | otherwise  = do
          n <- MaybeT $ reader $ lookup current
          (+1) <$> aux n (current : visited)
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

Note that our `Reader` monad shown in the previous chapter is quite different to the one defined in `transformers`. In fact, `Reader env a` is actually defined as `ReaderT env Identity a`. This is because it is generally quite uncommon to use the `Reader` monad by itself, since what it represents is just a plain function. The `ReaderT` monad transformer is defined as such:

```haskell
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
type Reader env a = ReaderT env Identity a
```

And the `Identity` monad is completely uninteresting:

```haskell
newtype Identity a = Identity { runIdentity :: a }
```

As such, the `transformers` library exposes some helper functions to make working with the plain `Reader` monad easier; for example, the `runReader` function extracts the enclosed function from a `ReaderT`, and the `reader` function transforms a function into a `ReaderT`.

We are done with our `Graph` library. Now, open `app/Main.hs` and write the following to see our `dfs` function in action (note that `print` is defined as `putStrLn . show`)!

```haskell
module Main where

import Data.Graph

myGraph :: Graph
myGraph = [(1, 2), (2, 3), (3, 1), (4, 5)]

main :: IO ()
main = do
  print $ dfs 1 2 myGraph
  print $ dfs 1 5 myGraph
```

We are done with developing our simple application! Compiling and running our program is simple with the help of build tools like `cabal`. In the terminal, just enter `cabal run` to compile the program (if changes have been made) and execute it!

```
> cabal run
Just 1
Nothing
```

LAST UPDATED **26 OCT 2024**

Monads in the Wild

Monads are so ubiquitous in programming that most libraries (even in general-purpose programming languages) expose monads. For example, the ReactiveX library in Java, which provides facilities for reactive programming, exposes an `Observable` class, which is a monad. In addition, most stream processors in data streaming libraries (across many languages) are also monads. You will typically know when something is a monad if it has a method called `flatMap`, which is the same as `>>=` in Haskell.

Therefore, whenever you are defining your own libraries for your own needs, think about what behaviours your library should support:

- Does your library involve nondeterminism or streams/lists of data?
- Does your library perform I/O?
- Does your library produce potentially empty computation?
- Does your library potentially fail?
- Does your library read from an environment?
- Does your library write to additional state?
- Does your library process state?

If the answer to one (or more) of the questions above is yes, chances are, your library should expose a monad! Furthermore, if you are writing Haskell code, your library functions can likely be described as the composition of some of the commonly used monads provided in Haskell's Prelude, the `transformers` library, or the `mtl` library.

Give this a try in the exercises and the assignment!

LAST UPDATED **26 OCT 2024**

In software engineering, the need to perform multiple tasks simultaneously gives rise to two (not mutually exclusive) approaches: concurrency and parallelism. For example, game servers are not monolithic entities; rather, a game server comprises many components, each interacting with the external world. One component might be dedicated to managing user chats, while several others process players' inputs and relay state updates back to them. Meanwhile, yet another component might be tasked with the complex calculations of game physics. This phenomenon is widely known as *concurrency*. Importantly, the successful operation of such a concurrent program doesn't necessarily rely on multiple processing cores, though their presence can certainly enhance performance and responsiveness.

On the other hand, *parallel* programs are typically centered around solving a single problem. For example, the act of summing the numbers in a large stream can be done sequentially; however, we prefer to split the large stream into smaller segments, and have one core dedicated to summing one segment, essentially allowing many cores to work in *parallel* to compute the main result. This is known as *parallelism*. Similarly, the functionality of a parallel program doesn't inherently depend on the availability of multiple cores.

Another key distinction between concurrent and parallel programs is how they engage with the outside world. By their very nature, concurrent programs are in constant interaction with networking protocols, databases, and similar systems. In contrast, a typical parallel program tends to be more focused in its operation. It streams in data, processes it intensively for a period, and then outputs the results, with minimal further I/O during that time.

The lines between concurrency and parallelism can often be blurred, particularly in traditional programming languages that compel developers to utilize the same constructs for both approaches.

In this chapter, we will see how functional programming concepts can be applied to concurrency and parallelism. For our course, assume that all our concurrent and parallel programs operate within the confines of a single OS process. We will then briefly look at some pitfalls of traditional concurrent and parallel programming, and see how purely functional languages tackle these.

LAST UPDATED **26 OCT 2024**

Concurrent Programming

In a usual *sequential* program, we do one thing completely after another. For example, if there are two tasks to do, `A` and `B`, then we will do `A` until completion, then do `B` until completion:

```
do A completely
do B completely
```

In a *concurrent* program, we do a little bit of either, arbitrarily:

```
work on A for 2ms
work on B for 1ms
work on A for 3ms
work on B for 6ms
work on A for 1ms
...
```

One of the advantages of writing concurrent programs (even in the presence of only a single operating system process) is that it would appear to the user that both tasks are executed simultaneously, making the program feel more fluid and have lower latency.

Typically, most programming languages provide building blocks for writing concurrent programs in the form of independent *threads of control*. Haskell is no exception. In Haskell, a thread is an I/O action that executes independently of other threads. To create a thread, we import the `Control.Concurrent` module and use the `forkIO` function. In the following example, we have one I/O action that writes `Hello World!` to a new file called `it5100a-notes.md`, and use `forkIO` to independently execute that I/O action in a separate thread immediately.

```
ghci> import Control.Concurrent
ghci> :t forkIO
forkIO :: IO () -> IO ThreadId
ghci> import System.Directory
ghci> forkIO (writeFile "it5100a-notes.md" "Hello World!") >> doesFileExist
"it5100a-notes.md"
False
```

The Haskell runtime does not specify an order in which threads are executed. Thus, the file `it5100a-notes.md` created by the new thread may or may not have been created by the time the original thread checks for its existence. If we try this example once, and then remove `it5100a-notes.md` and try again, we may get a different result the second time. In general, concurrent programs are *nondeterministic*.

From earlier, we stated that concurrent programs can "hide" the latency of programs by executing multiple tasks concurrently. This makes applications more responsive. For example, a web browser needs to process user input (like button clicks etc.) and page loads or running JavaScript processes. If a web browser were programmed sequentially, then the page must load completely before the user can interact with the browser at all. In addition, JavaScript processes will usually be running in the background, and while it is doing so, the user cannot interact with the browser either. However, since web browsers are (almost always) concurrent, the user can continue to interact with them while background processes are running, even if the browser is only running on a single CPU core.

A toy example demonstrating this is as follows. We shall write a program that receives some user input and creates a large file with user specified contents:

```
import Data.List

writeContents :: String -> String -> IO ()
writeContents file_name contents = do
  let c = intercalate "\n" $ replicate 1000000 (intercalate "" $ replicate 100
contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"

main :: IO ()
main = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          forkIO $ writeContents x y
          main
```

Observe the `main` I/O action. All `main` does is to read user input. The logic for writing the user input to a file is done by `writeContents`, which is done on a separate thread. This way, `main` can read user input immediately after `writeContents` is forked, and is ready to read more user input again, without having to wait for `writeContents` to complete its task first. If we hadn't used `forkIO`, `main` would perform the writing on the same thread completely, which may take a while, before being able to read user input again.

Communication Between Threads

The simplest way to share information between two threads is to let them both use a variable. In our file generation, the main thread shares both the name of a file and its contents with the other thread. Because Haskell data is immutable by default, this poses no risks: neither thread can modify the other's view of the file's name or contents. However, we will often need to have threads actively communicating with each other. For example, GHC

does not provide a way for one thread to emit data to another thread, or let another thread know that it is still executing or has terminated.

MVars

The way we do this in Haskell is to use a synchronizing variable called the `MVar`. An `MVar` is essentially a mutable variable holding a value. You can put something in a variable making it full, and take out the value from a full `MVar`, making it empty.

```
ghci> import Control.Concurrent.MVar
ghci> :t putMVar
putMVar :: MVar a -> a -> IO ()
ghci> :t takeMVar
takeMVar :: MVar a -> IO a
```

Importantly, using `putMVar` on a full `MVar` causes the thread to *block* until the `MVar` becomes empty; dually, using `takeMVar` on an empty `MVar` causes the thread to *block* until the `MVar` becomes full.

An example of using `MVar`s is as follows. We are going to use the same toy example from earlier, except we are going to ensure that only one thread can perform file writing. This is so that we do not hog computing resources from other parts of our system (which there aren't, but suppose there are):

```
import Data.List
import Control.Concurrent.MVar

writeContents :: String -> String -> MVar () -> IO ()
writeContents file_name contents lock = do
  takeMVar lock
  putStrLn $ "Write to " ++ file_name ++ " started"
  let !c = intercalate "\n" $ replicate 1000000 (intercalate "" $ replicate 500
contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
  putMVar lock ()

mainLoop :: MVar () -> IO ()
mainLoop lock = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then do takeMVar lock
          return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          forkIO $ writeContents x y lock
          mainLoop lock

main :: IO ()
main = do
  lock <- newMVar ()
  mainLoop lock
```

Upon executing this program, the `main` I/O action initializes a single `MVar` which is then passed onto other parts of the program. The `mainLoop` action does the same as before, except that it receives the `lock` from `main`. Note the `then` branch in `mainLoop` —if the user enters `exit`, it waits for the `lock` to be filled with a value, signalling the completion of file writing, before exiting the program. Importantly, `takeMVar` wakes up in FIFO order. In other words, we are guaranteed that no more threads of `writeContents` will be waiting to be executed at this point, because `takeMVar` will only wake up the `mainLoop` thread once all earlier `writeContents` threads have been executed. The `writeContents` action performs the actual file writing as per usual; however, it first acquires the shared `lock` before executing the file writing operation, before putting back the `lock` value. This is so that only one thread can perform `writeContents` at any time.

Channels

Aside from `MVar`s, we can also provide one-way communication channels via the `Chan` type. Threads can write to channels (without blocking), and can read from channels (blocking if the channel is empty):

```
ghci> import Control.Concurrent.Chan
ghci> :t writeChan
writeChan :: Chan a -> a -> IO ()
ghci> :t readChan
readChan :: Chan a -> IO a
```

An example is as follows:

```
import Data.List
import Control.Concurrent.Chan

writeContents :: Chan String -> IO ()
writeContents chan = do
  file_name <- readChan chan
  contents  <- readChan chan
  putStrLn $ "Write to " ++ file_name ++ " started"
  let !c = intercalate "\n" $ replicate 1000000 (intercalate "" $ replicate 500
contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
  writeContents chan

mainLoop :: Chan String -> IO ()
mainLoop chan = do
  putStrLn "Enter filename:"
  x <- getLine
  if x == "exit"
  then return ()
  else do putStrLn "Enter file contents:"
          y <- getLine
          writeChan chan x
          writeChan chan y
          mainLoop chan

main :: IO ()
main = do
  chan <- newChan :: IO (Chan String)
  forkIO $ writeContents chan
  mainLoop chan
```

In this example, only two threads are ever spawned: (1) the main thread which runs `main` and `mainLoop`, which, like before, only reads user input and passes the file name and contents to the other thread, which performs file writing. `main` initializes the channel and forks another thread to `writeContents`. `mainLoop` receives user input and passes them through the channel, which is read by `writeContents` for file writing.

More Cores!

Thus far, we have only discussed how the Haskell runtime is able to spark new threads of control. As said before, this does not guarantee that the program is actually running on

multiple cores. In fact, by default, the program only runs on a single core. We can inspect this by using the `numCapabilities` function:

```
import GHC.Conc
main :: IO ()
main = print numCapabilities
```

```
> ghc Main.hs
> ./Main
1
```

There is in fact, a way to set the number of CPU cores being used by the Haskell runtime. This can be done using the `setNumCapabilities` function.

```
import Control.Concurrent
main :: IO ()
main = do setNumCapabilities 4
          print numCapabilities
```

However, compiling and running this program gives a warning:

```
ghc Main.hs
./Main
Main: setNumCapabilities: not supported in the non-threaded RTS
1
```

The reason for this is because Haskell uses two runtime systems: (1) a non-threaded runtime, and (2) a threaded runtime. By default, compiling our program with `ghc` links the non-threaded runtime, which is not able to leverage multiple cores. Therefore, if we want to use multiple cores, we have to use the threaded runtime instead. This can be done by compiling the program with the `-threaded` option.

```
ghc Main.hs -threaded
./Main
4
```

Another way to specify the number of cores being used is to provide the `+RTS -Nx` option, where `x` is the number of cores we would like to use.

If we are using `cabal` to build our project, we can provided `-threaded` as a GHC option via the `ghc-options` setting:

```
executable playground
    -- ...
    ghc-options: -threaded
```

And execute it with `cabal run -- +RTS -Nx`.

Let's give this a try!

```
import Data.List
import Control.Concurrent

writeContents :: String -> String -> Chan () -> IO ()
writeContents file_name contents chan = do
  putStrLn $ "Write to " ++ file_name ++ " started"
  let !c = intercalate "\n" $ replicate 1000000 (intercalate "" $ replicate 500
contents)
  writeFile file_name c
  putStrLn $ file_name ++ " written"
  writeChan chan ()

main :: IO ()
main = do
  n <- getNumCapabilities
  putStrLn $ "Number of cores: " ++ show n
  chan <- newChan :: IO (Chan ())
  forkIO $ writeContents "abc" "def" chan
  forkIO $ writeContents "def" "ghi" chan
  _ <- readChan chan
  _ <- readChan chan
  return ()
```

This program is simple. The `main` I/O action creates a channel and passes them to two threads that perform file writing. Once each thread has completed writing the file, they will also write to the channel, signalling completion. The `main` I/O action will only terminate once both threads have completed.

Let's try timing our program with different runtime options. The first execution command runs our program with 4 cores, while the second one only uses 1. We use the `time` shell command to time the execution of each program:

```
> time cabal run playground -- +RTS -N4
Number of cores: 4
Write to abc started
Write to def started
abc written
def written

------------------------------------------------------
Executed in    6.44 secs    fish           external
   usr time   10.76 secs  379.00 micros   10.76 secs
   sys time    1.15 secs  137.00 micros    1.15 secs
```

```
> time cabal run
Number of cores: 1
Write to def started
Write to abc started
def written
abc written


------------------------------------------------------
Executed in   12.02 secs    fish           external
   usr time   10.78 secs    0.00 micros   10.78 secs
   sys time    0.84 secs  560.00 micros    0.84 secs
```

Notice that the `usr` and `sys` times for both are roughly similar. This is not surprising, because `usr` and `sys` times reflect CPU execution time; loosely, if the CPU spends 1s executing on one core and 1s executing on another core, then the total of `usr` and `sys` time reported with be 2s. This is to be expected, because the same amount of work needs to be done to write to our files. However, what we really want to profile is the *real* or *wall clock* time, i.e. how much time had elapsed on the clock. As you can see, the multicore execution ran roughly 2x faster than the single core execution, since we have two files we can write in parallel!

LAST UPDATED 26 OCT 2024

Parallel Programming

Let us now turn our focus to parallel programming. For many large problems, we could divide them into chunks and evaluate the solution for these chunks at the same time on multiple cores, before combining the results, just like a divide-and-conquer approach. However, doing so is traditionally seen as difficult, and we usually use the same libraries and language primitives that are used for concurrency to develop a parallel program. Writing parallel programs in general-purpose imperative languages can be complex and tedious.

While we could certainly use Haskell's concurrency features like `forkIO`, `MVar` and `Chan` to develop parallel code, there is a much simpler approach available to us. All we need to do is to annotate some sub-expressions in our functions to make them evaluated in parallel.

Non-Strict Evaluation

In the very beginning of this course, we described Haskell as a non-strict evaluation language. That is, Haskell decides the evaluation strategy for us, unlike other strict evaluation languages where things are evaluated in a deterministic and specific format. For example, in Python, a function call is evaluated by first fully evaluating its arguments, then executing each statement in the function from top down. Haskell generally only evaluates terms *by need*, giving rise to a notion of *lazy evaluation*.

The key idea of attaining *parallelism* in Haskell is by specifying *parallel evaluation strategies*.

Strict Evaluation

Before we bgin describing how to evaluate terms in parallel, we must first describe how we can even force the evaluation of a term in the first place. For example, in the following program:

```
ghci> x = [1..]
ghci> y = sum x
```

virtually nothing is evaluated, and GHCI does not enter an infinite loop. This is because there is as yet no demand for the evaluation of `y`. Of course, if we attempt to evaluate `y`, we do arrive at an infinite loop, because evaluating the actual sum of `x` is required to determine what `y` is.

Therefore, whenever an expression is encountered, Haskell allocates a *thunk* as a uncomputed placeholder for the result of the expression evaluation. The thunk is only evaluated by need (usually as little as possible) to evaluate other parts of code.

For example:

```
ghci> x = [1..]
ghci> case x of { [] -> 0; (x:xs) -> x }
1
```

Notice that the `case` expression demands the evaluation of `x`. However, it does not demand the *complete* evaluation of `x`. Instead, it only demands to know the constructor of `x`. Therefore, when executing `x = [1..]`, Haskell puts a completely unevaluated thunk, for `x`, and the `case` expression then evaluates `x` to *head normal form* (HNF) (evaluating to the constructor but not its arguments)[1] to perform the case analysis.

Another example of lazy evaluation is with `let` expressions:

```
ghci> let x = [1..]; y = sum x in 1 + 2
3
```

Again, Haskell does not evaluate `y` at all since it is not demanded in the evaluation of `1 + 2`!

This may be a problem for concurrency and parallelism, because it is possible for `forkIO` to push an I/O action to a different thread, only for that thread to allocate an unevaluated thunk for it, and when its evaluation is demanded, the evaluation is done on the main thread!

```
expensive :: MVar String -> IO ()
expensive var = do
    putMVar var expensivelyComputedString

main :: IO ()
main = do
    var <- newEmptyMVar
    forkIO $ expensive var
    whatever
    result <- takeMVar var
    print result
```

The program above gives the impression that the expensive computation is done on the forked thread. However, in reality, what could happen is that the thread running `expensive` only allocates a thunk for `expensivelyComputedString`, and returns. Then, when the `result` is demanded in the `main` I/O action running in the main thread, it is the main thread that computes the expensively computed string, thereby, achieving nothing from the concurrency.

It is for this reason that Haskell exposes primitives for deciding the evaluation of expressions. The one most used is `seq`, which introduces an artificial demand for an expression to be evaluated to head normal form:

```
ghci> :t seq
seq :: a -> b -> b
```

The expression `x `seq` y` evaluates to `y`, but creates an artificial demand for the evaluation of `x` as well. Therefore, evaluating the following expression does not terminate:

```
ghci> let x = [1..]; y = sum x in y `seq` 1 + 2
```

However, notice that the following *does* terminate:

```
ghci> let x = [1..] in x `seq` 1 + 2
3
```

This is because `seq` only creates an artificial demand for `x` to be evaluated to *head normal form*, i.e. up to the evaluation of its constructor.

What we can do instead is to introduce a new *evaluation strategy* for forcing the full evaluation of a list:

```
ghci> :{
ghci| deepSeq :: [a] -> b -> b
ghci| deepSeq [] x = x
ghci| deepSeq (x:xs) y = x `seq` deepSeq xs y
ghci| :}
ghci> x = [1..]
ghci> x `seq` 1
1
ghci> x `deepSeq` 1
```

Using `deepSeq` now forces the full evaluation of `x`, which obviously does not terminate because `x` is infinitely large! However, note that `deepSeq` only evaluates the *elements* to HNF—therefore, if `x` were a, for example, a two-dimensional list, the individual one-dimensional lists in `x` are only evaluated to HNF, i.e. only their constructors are evaluated.

Parallel Evaluation

Since parallel programming is all about deciding what expressions to evaluate in parallel, all we need is some primitives that tell the compiler to evaluate an expression in parallel, just like `seq`! The `GHC.Conc` module exposes two evaluation primitives, `par` and `pseq` that allows us to do parallel programming easily:

```
ghci> import GHC.Conc
ghci> :t par
par :: a -> b -> b
ghci> :t pseq
pseq :: a -> b -> b
```

`par` is straightforward to understand: `x `par` y` is an expression stating that there is an artificial demand for `x` that *could* be evaluated to HNF in *parallel*. However, `par` does not *guarantee* the parallel evaluation of `x`. This is because `x` could be a cheap computation that does not need to be, and should not be, evaluated in parallel, or that there are not enough cores available for the parallel evaluation of `x`.

Then, what is `pseq` for? Notice this: in an expression `x `par` f x y`, we claim to want to evaluate `x` in parallel to HNF, *and then* combine it with `y` using `f` in the current thread. However, this requires a guarantee that `y` is evaluated on the current thread *before* the current thread attempts to evaluate `x`. Otherwise, it could be that `par` will queue a spark for the evaluation of `x`, and before a new thread can be sparked for that evaluation, the current thread evaluates `f x y`, which performs the evaluation of `x` first; therefore, no parallel evaluation of `x` happens, defeating of `par` in the first place.

Therefore, we need some primitive that performs the evaluation of an expression to HNF before another expression. `seq` does not do this; `x `seq` y` only claims to evaluate `x` to HNF, but does not enforce that to happen *before* `y`. In contrast, `pseq` does. `x `pseq` y` guarantees that the evaluation of `x` to HNF happens *before* the evaluation of `y`.

As such, `par` and `pseq` allow us to annotate computations with evaluation strategies to describe what computation happens in parallel, and what that computation is *in parallel with*. For example, the expression `x `par` (y `pseq` f x y)` states roughly that `x` happens in parallel with `y`, then the results are combined using `f`.

For example, let us try writing a parallel (but still exponential) fibonacci:

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = n1 `par` (n2 `pseq` (n1 + n2))
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
```

Aside from the usual base cases, the recursive case computes the $n-1$ and $n-2$ fibonacci numbers in parallel, then combines them together with addition. Described in words, the recursive case computes `fib (n - 1)` in parallel with `fib (n - 2)` by queueing a spark for `fib (n - 1)` and evaluating `fib (n - 2)` in the current thread, then adds the results together with plain addition.

Computing `fib 45` shows that for large values, having more cores makes a big difference.

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
1134903170

------------------------------------------------------
Executed in    3.29 secs    fish           external
   usr time   53.14 secs  319.00 micros   53.14 secs
   sys time    0.47 secs  129.00 micros    0.47 secs



> time cabal run playground -- +RTS -N1
Number of cores: 1
1134903170

------------------------------------------------------
Executed in   12.93 secs    fish           external
   usr time   12.61 secs  418.00 micros   12.61 secs
   sys time    0.08 secs  171.00 micros    0.08 secs
```

When Should We Parallelize?

However, notice the `usr` time for the case of running our program on 20 cores. Clearly, the CPU does more than 4x more work than the single core case; it just so happens that leveraging more cores makes the speed-ups outweigh the additional overhead. Indeed, while `par` is cheap, it is not *free*. Although Haskell threads are lightweight, threads in general will always incur some additional overhead, and at some point, the benefits of computing something in parallel are outweighed by the overhead of spawning a new thread for its computation. For example, in the case of computing `fib 3`, it is frankly completely unnecessary to compute `fib 2` and `fib 1` in parallel, since both are such small computations that run incredibly quickly.

Let us amend our implementation to only use parallelism for larger values. Smaller values are computed sequentially:

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
-- sequential for small n
fib n | n <= 10 = fib (n - 1) + fib (n - 2)
-- parallel for large n
fib n = n1 `par` (n2 `pseq` (n1 + n2))
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
```

The execution time shows a significant speed-up on both the single core and multicore runtimes!

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
1134903170

------------------------------------------------------
Executed in  892.37 millis    fish           external
   usr time   13.01 secs    646.00 micros   13.00 secs
   sys time    0.18 secs      0.00 micros    0.18 secs



> time cabal run playground -- +RTS -N1
Number of cores: 1
1134903170

------------------------------------------------------
Executed in    6.81 secs    fish           external
   usr time    6.71 secs  453.00 micros    6.71 secs
   sys time    0.03 secs    0.00 micros    0.03 secs
```

Generally speaking, knowing when to parallelize is a matter of experimentation, trial-and-error and engineering experience. It highly depends on the computation you are trying to parallelize, the kind of computation you are doing, the usual inputs to the computation, and so on.

Parallel Strategies

Let us try writing a parallel mergesort:

```haskell
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort ls
  | n < 100 = merge left' right'
  | otherwise = par left' $ pseq right' $ merge left' right'
    where n = length ls `div` 2
          merge [] ys = ys
          merge xs [] = xs
          merge (x:xs) (y:ys)
            | x <= y = x : merge xs (y : ys)
            | otherwise = y : merge (x:xs) ys
          (left, right) = splitAt n ls
          left' = mergesort left
          right' = mergesort right
```

Our `mergesort` function does a typical merge sort, except from the fact that we are using an immutable list. Let us write a supporting `main` function to test our program:

```haskell
main :: IO ()
main = do
  n <- getNumCapabilities
  putStrLn $ "Number of cores: " ++ show n
  let ls :: [Int] = [10000000, 9999999..1]
      ls' = mergesort ls
  print $ length ls'
```

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
10000000

------------------------------------------------------
Executed in    3.58 secs    fish           external
   usr time   16.02 secs  381.00 micros   16.02 secs
   sys time    1.39 secs  159.00 micros    1.39 secs


> time cabal run playground -- +RTS -N1
Number of cores: 1
10000000

------------------------------------------------------
Executed in    6.11 secs    fish           external
   usr time    5.62 secs    0.00 micros    5.62 secs
   sys time    0.43 secs  586.00 micros    0.43 secs
```

From before, recall that because Haskell is a lazy language, it may be the case that not all the supposedly parallel computation happens in the other thread. Since both `par` and `pseq` evaluate their first arguments only to HNF, it really only does evaluation up until it determines the constructor of the list after sorting, leaving the remainder of the list unevaluated. Then, in `main`, when we obtain the `length` of the list, the main thread may then have to evaluate the remainder of the list in the same thread. Let us extract some more performance out of our parallel evaluation by actually evaluating everything deeply in the parallel computation using `deepSeq` from before:

```haskell
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort ls
  | n < 100 = merge left' right'
  | otherwise = par (deepSeq left') $ pseq right' $ merge left' right'
    where n = length ls `div` 2
          merge [] ys = ys
          merge xs [] = xs
          merge (x:xs) (y:ys)
            | x <= y = x : merge xs (y : ys)
            | otherwise = y : merge (x:xs) ys
          (left, right) = splitAt n ls
          left' = mergesort left
          right' = mergesort right

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

Now we should notice some more performance gains!

```
> time cabal run playground -- +RTS -N20
Number of cores: 20
10000000

------------------------------------------------------
Executed in    2.89 secs    fish           external
   usr time   18.04 secs  365.00 micros   18.04 secs
   sys time    0.68 secs  145.00 micros    0.67 secs


> time cabal run playground -- +RTS -N1
Number of cores: 1
10000000

------------------------------------------------------
Executed in    6.18 secs    fish           external
   usr time    5.59 secs  362.00 micros    5.59 secs
   sys time    0.46 secs  145.00 micros    0.46 secs
```

Some very smart people have also come up with nice and elegant ways to write parallel code. For example, using the `parallel` library, we can express parallel programs with `Strategy`'s in the `Eval` monad:

```haskell
mergesort :: (Ord a, NFData a) => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort ls
  | n < 100 = merge left' right'
  | otherwise = runEval $ do
      l <- rparWith rdeepseq left'
      r <- rseq right'
      return $ merge l r
  where n = length ls `div` 2
        (left, right) = splitAt n ls
        left' = mergesort left
        right' = mergesort right
        merge [] ys = ys
        merge xs [] = xs
        merge (x:xs) (y:ys)
          | x <= y = x : merge xs (y : ys)
          | otherwise = y : merge (x:xs) ys
```

Strategies also allow us to separate algorithm from evaluation. For example, we can write a parallel fibonacci like so:

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n | n <= 10 = fib (n - 1) + fib (n - 2)
fib n = runEval $ do
    n1 <- rpar (fib (n - 1))
    n2 <- rseq (fib (n - 2))
    return $ n1 + n2
```

Alternatively, we can make clear the distinction between the underlying algorithm and the evaluation strategy with `using`:

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n | n <= 10   = n1 + n2
      | otherwise = (n1 + n2) `using` strat
  where n1 = fib (n - 1)
        n2 = fib (n - 2)
        strat v = do { rpar n1; rseq n2; return v }
```

We will leave it up to you to learn more about parallel Haskell with the `parallel` library. For more information, you may read the paper by Marlow et al.; 2010 that describes it. We shall not cover these because they, along with `par` and `pseq`, are much more Haskell-specific and less applicable to code written in general-purpose languages. The only goal of this chapter, which we hope has been achieved, is to show how easy it is to introduce parallelism to regular sequential programs in a purely functional programming language.

---

[1] Usually expressions are evaluated to *weak head normal form* (WHNF), although the distinction is not crucial for our understanding.

References

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. 2010. `seq` No More: Better Strategies for Parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell (Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/1863523.1863535.

LAST UPDATED **26 OCT 2024**

Software Transactional Memory

Concurrency and parallelism is, generally, *really hard*. This is because the ordering of concurrent and parallel evaluation is *nondeterministic*, and the traditional threaded model of concurrent programming with threads make working with concurrent operations and *composing* them is very difficult and error-prone. `MVar`s and Haskell's runtime make this slightly safer (if you have done concurrency in other languages like C before, you might be able to see why), but are still vulnerable to the same issues that plague concurrent and parallel programs.

To give you a toy example, suppose we have two threads, one which acquires two `MVar`s `a` and `b` and adds the value of `a` to `b`, and another which swaps their values. One possible implementation might be the following, with a deadly vulnerability hidden in plain sight:

```
swap :: MVar a -> MVar a -> Chan () -> IO ()
swap a b chan = do
    x <- takeMVar a
    y <- takeMVar b
    putMVar a y
    putMVar b x
    writeChan chan () -- signal done

addToMVar :: Num a => MVar a -> MVar a -> Chan () -> IO ()
addToMVar a b chan = do
    y <- takeMVar b
    x <- takeMVar a
    let z = x + y
    putMVar b z
    putMVar a x
    writeChan chan () -- signal done

main :: IO ()
main = do
    a <- newMVar 1 :: IO (MVar Int)
    b <- newMVar 2 :: IO (MVar Int)
    chan <- newChan :: IO (Chan ())
    forkIO $ addToMVar a b chan
    forkIO $ swap a b chan
    _ <- readChan chan
    _ <- readChan chan
    x <- takeMVar a
    y <- takeMVar b
    print x
    print y
    return ()
```

In this program, several things could happen:

- `swap` starts first, and is able to acquire the values from both `MVar`s `a` and `b`, thus executing completely and putting new values to `a` and `b` for `addToMVar` to use
- `addToMVar` starts first and is able to acquire the values from both `MVar`s `a` and `b`, thus executing completely and putting new values to `a` and `b` for `swap` to use
- `swap` starts first and acquires `a`, shortly thereafter `addToMVar` begins and acquires `b`. Now `swap` is waiting for `b`, and `addToMVar` is waiting for `a`.
- `addToMVar` starts first and acquires `b`, shortly thereafter `swap` begins and acquires `a`. Now `swap` is waiting for `b`, and `addToMVar` is waiting for `a`.

The last two scenarios result in something known as a *deadlock* and causes all these threads to wait and to be unable to continue. In particular, this deadlock was caused by a *lock ordering inversion*, a very common mistake that is usually undetectable by the compiler, and only starts causing problems at runtime! Scenarios like these are known as *race conditions*, and yes, while there are tools to detect *race conditions*, detecting *all* race conditions is *undecidable*, and thus is an impossible problem to solve. Are there tools to help us reduce of likelihood of running into race conditions?

Haskell supports something known as *software transactional memory* (STM) (Harris et al.; 2005), which is very similar to *transactions* in databases with ACID guarantees. Notice that this deadlock situation could go away if `swap` and `addToMVar` acquired both locks in one atomic operation, so that neither thread can interleave an `MVar` acquisition! STM provides such facilities to allow us to define, compose and work with atomic transactions. All we need to do is to install the `stm` package!

Key Ideas

Instead of working with the `IO` monad, STM constructs work within the `STM` monad. Under the hood, the `stm` implementation handles all the coordination, so as programmers, as long as we are working within the `STM` monad, we can regard these operations as atomic. In other words, an STM transaction appears to take place indivisibly. All transactional operations are within `STM`, and can only be escaped to `IO` using `atomically`:

```
ghci> import Control.Concurrent.STM
ghci> :t atomically
atomically :: STM a -> IO a
```

An `atomically` block is treated as a single I/O operation, so the `STM` operations cannot interleave. In addition, the `atomically` block executes a transaction entirely, or not at all.

Now let us try using `STM` for communications between threads. We are going to create a transaction for atomically acquiring both `MVar`s. Of course, instead of `MVar`, which operates in the `IO` monad, the `Control.Concurrent.STM` module exposes a `TMVar`, sort of like a *transactional* `MVar` that lives in the `STM` monad. Let us write this transaction:

```haskell
takeBothTMVars :: TMVar a -> TMVar b -> STM (a, b)
takeBothTMVars a b = do
  x <- takeTMVar a
  y <- takeTMVar b
  return (x, y)
```

As you can see, this looks just like an equivalent version written for `MVar` s:

```haskell
takeBothMVars :: MVar a -> MVar b -> IO (a, b)
takeBothMVars a b = do
  x <- takeMVar a
  y <- takeMVar b
  return (x, y)
```

Now let us rewrite our original deadlocked program using `TMVar` s and `STM` , focusing temporarily on the `swap` function. Recall that we want to take both `TMVar` s as a single atomic operation, hence we defined an `STM` operation `takeBothTMVars` that does so. To actually perform this operation as a single I/O action, we have to use `atomically` which performs the transaction, atomically:

```haskell
swap :: TMVar a -> TMVar a -> Chan () -> IO ()
swap a b chan = do
    (x, y) <- atomically $ takeBothTMVars a b
    -- ...
```

This way, the transaction is done in one fell swoop, and if either `TMVar` s are empty, the thread running `swap` will block until both become available. We can do the same for `addToMVar` , but this time, we are going to introduce lock-order inversion again by swapping the arguments to `takeBothTMVars` :

```haskell
addToMVar :: Num a => TMVar a -> TMVar a -> Chan () -> IO ()
addToMVar a b chan = do
    (y, x) <- atomically $ takeBothTMVars b a
```

Although we swapped the arguments to `takeBothTMVars` , thereby introducing lock-order inversion, operationally, there is no difference, since `takeBothTMVars` is regarded as a single atomic operation anyway. We then continue defining the rest of the program which should be similar to before. Importantly, note that to create a new `TMVar` within `IO` for coordination, we use the `newTMVarIO` function;

```haskell
import Control.Concurrent
import Control.Concurrent.Chan
import Control.Concurrent.STM

takeBothTMVars :: TMVar a -> TMVar b -> STM (a, b)
takeBothTMVars a b = do
  x <- takeTMVar a
  y <- takeTMVar b
  return (x, y)

putBothTMVars :: TMVar a -> a -> TMVar b -> b -> STM ()
putBothTMVars a x b y = do
  putTMVar a x
  putTMVar b y

swap :: TMVar a -> TMVar a -> Chan () -> IO ()
swap a b chan = do
    (x, y) <- atomically $ takeBothTMVars a b
    atomically $ putBothTMVars a y b x
    writeChan chan ()

addToMVar :: Num a => TMVar a -> TMVar a -> Chan () -> IO ()
addToMVar a b chan = do
    (y, x) <- atomically $ takeBothTMVars b a
    let z = x + y
    atomically $ putBothTMVars a x b z
    writeChan chan ()

main :: IO ()
main = do
    a <- newTMVarIO 1 :: IO (TMVar Int)
    b <- newTMVarIO 2 :: IO (TMVar Int)
    chan <- newChan :: IO (Chan ())
    forkIO $ addToMVar a b chan
    forkIO $ swap a b chan
    _ <- readChan chan
    _ <- readChan chan
    x <- atomically $ takeTMVar a
    y <- atomically $ takeTMVar b
    print x
    print y
```

We don't have to only use STM for coordination between threads (although that is certainly a great use case). As long as we want atomic memory transactions, it is highly likely that STM is applicable.

For example, suppose we have some in-memory shared state, such as a counter, and users (perhaps across the network) can modify this counter. Modifying the counter requires two things: (1) reading the existing counter, (2) modifying the read value, (3) updating the counter with the modified value. To prevent data races, we want all these operations to be done in one fell swoop (i.e. as a single transaction).

```
incVar :: TVar Int -> STM Int
incVar v = do
  x <- readTVar v
  let y = x + 1
  writeTVar v y
  return y
```

Now we're not afraid to compose `incVar` with other STM operations, even if they are done concurrently!

```
import Control.Concurrent
import Control.Concurrent.STM

-- Increments a 'TVar'
incVar :: TVar Int -> STM Int
incVar v = do
  x <- readTVar v
  let y = x + 1
  writeTVar v y
  return y

-- IO Action that increments a TVar five times
aIncVar :: TVar Int -> IO ()
aIncVar v = aux 5 where
  aux :: Int -> IO ()
  aux 0 = return ()
  aux n = do
    r <- atomically $ incVar v
    print r
    aux (n - 1)

main :: IO ()
main = do
    n <- getNumCapabilities
    putStrLn $ "Number of cores: " ++ show n
    -- Initialize the counter
    counter <- newTVarIO 0 :: IO (TVar Int)
    -- For example, run four threads that increment the counter 5 times
    forkIO $ aIncVar counter
    forkIO $ aIncVar counter
    forkIO $ aIncVar counter
    forkIO $ aIncVar counter
    -- Sleep so we can wait for the other threads to complete
    threadDelay 1000000
```

When executing this program, you should notice that the counter is being incremented correctly, with a final value of `20`.

The `stm` library provides many other useful facilities for writing transactional programs. Refer to the library documentation or the original paper for more details.

Concurrent and Parallel Programming in Haskell

In summary, concurrent and parallel programming in Haskell is, generally, not too dissimilar to that in other general-purpose languages. However, because Haskell is a purely functional and non-strict evaluation language, there are several neat things at our disposable. For one, it is relatively straightforward to fork an I/O action to be performed concurrently, and to use synchronizing variables like `MVar` for communication between threads. Importantly, the Haskell runtime ensures that `MVar` s are only taken from or put by one thread, so synchronization is inherent in its implementation. However, using `MVar` s alone can get cumbersome especially when dealing with multiple concurrent operations that do not compose well; hence, the introduction of STM for atomic transactions to reduce the likelihood of accidentally introducing race conditions and deadlocks. In addition, because Haskell has non-strict evaluation, parallelizing it is fairly straightforward, by simply annotating the functions with `par` and `pseq` function applications to describe what operation should be done in parallel with what else.

Most importantly, ideas like I/O actions, STM transactions and even parallel evaluation strategies are all exposed as monads, and programs that are written with these can make use of all the guarantees and conveniences that monads have to offer. As before, monads are some of the most powerful concepts in programming, and it helps dramatically to have programming languages that make working with them easy.

Lastly, concurrency and parallelism are huge topics in Computer Science in and of itself. Since many of what is described in this course are not as generally applicable to other general-purpose languages, many of the details are omitted. More information is readily available online and in the original papers describing the various systems like Concurrent and Parallel Haskell, and STM. This may be useful if you are interested in a pursuing a career involving Haskell development, or wish to learn, more deeply, about some of the ideas we have presented.

References

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952.

LAST UPDATED **26 OCT 2024**

I highly recommend that you work through the exercises *before* looking at the worked solutions!

Course Introduction

Question 1

1. `17`. $(3 \times 4) + 5 = 12 + 5 = 17$.
2. `23`. $3 + (4 \times 5) = 3 + 20 = 23$. Note that `*` has higher precedence than `+`.
3. `1`. Exponentiation has a higher precedence than modulo (non-operator functions like `mod` that are called in an infix manner can have a well-defined operator precedence level).
4. `24.25`. Regular division of integers gives a `Fractional` type.
5. `24`. The `div` function is similar to `//` in Python.
6. `1`. First we evaluate the condition `let x = 3 in x + 3` evaluates to `3 + 3` which therefore is `6`. Clearly `6 /= 5` is true, so we need to also evaluate `3 < 4`, which is also true. `&&` is the same as `and` in Python, so `True and True` is therefore `True`. Thus, the whole expression evaluates to the if branch, which is `1`.
7. `False`. `otherwise` is actually just `True` by definition, so `not True` becomes `False`.
8. It actually causes a compile-time error since it is a type error. `fst` and `snd` receive pairs, so these functions do not work on triples.
9. `1.5`. The `succ` function returns the successor of any enumerable type. For numbers, this would be one more than the number.
10. `1.4142135623730951`. Straightforward. Notice that Haskell's Prelude (the built-in stuff) comes with many math functions.
11. `True`. The `elem` function is similar to `in` in Python.
12. `4`. When writing `let` bindings in a single line, we can separate multiple definitions with `;`. Therefore, we have defined two functions `f` and `g` which add one and multiply by 2 respectively. The `.` operator is function composition, where $(g \circ f)(x) = g(f(x))$, so `(g . f) 1` is the same as `g (f 1)`, which evaluates to `4`.
13. `[1, 2, 3, 4, 5, 6]`. This is straightforward, since `++` concatenates two lists.
14. `1`. `head` returns the first element of the list.
15. `[2, 3]`. `tail` returns the suffix of the list without the first element.

16. `[1, 2]. init` returns the list without the last element.
17. `1. !!` is indexing.
18. `True. null` checks whether a list is empty.
19. `3`. Obvious.
20. `[3]. drop n` drops the first `n` elements of a list.
21. `[-1, 0, 1, 2, 3]. take n` takes the first `n` elements of a list. The range `[-1..]` is an infinitely long range from `-1` to infinity.
22. `[5, 1, 2 ,3]. dropWhile f` will drop elements from a list until `f` returns false for an element.
23. `30`. The easiest way to see this is by converting this to the equivalent Python expression:

```
>>> sum([x[0] for x in
            [(i, j) for i in range(1, 5)
                    for j in range(-1, 2)]])
```

Going back to Haskell land, let us evaluate the inner list first. `[(i, j) | i <- [1..4], j <- [-1..1]]` gives `[(1, -1), (1, 0), (1, 1), (2, -1), ..., (4, 1)]` then, `[fst x | x <- ...]` would therefore give `[1,1,1,2,2,2,3,3,3,4,4,4]` which sums to 30.

Question 2

Idea: take the last elements of both lists, and check for equality. For this, we can use the `last` function.

```
eqLast xs ys = last xs == last ys
```

Question 3

Idea: reverse the string, and check if the string and its reverse are equal. For this, we can use the `reverse` function.

```
isPalindrome w = w == reverse w
```

Question 4

```haskell
taxiFare f r d = f + r * d
```

Question 5

There are several ways to approach this problem. Let us first define the `ingredientPrice` function which should be straightforward to do.

```haskell
ingredientPrice i
  | i == 'B' = 0.5
  | i == 'C' = 0.8
  | i == 'P' = 1.5
  | i == 'V' = 0.7
  | i == 'O' = 0.4
  | i == 'M' = 0.9
```

Then we can define `burgerPrice` recursively. If the string is empty then the price is 0. Otherwise, take the price of the first ingredient and add that to the price of the remaining burger.

```haskell
burgerPrice burger
  | null burger = 0
  | otherwise =
      let first = ingredientPrice (head burger)
          rest  = burgerPrice (tail burger)
      in  first + rest
```

Of course, we know that we can do the following in Python quite nicely:

```python
def burger_price(burger):
    return sum(ingredient_price(i) for i in burger)
```

This can be done in Haskell too as follows:

```haskell
burgerPrice burger = sum [ingredientPrice i | i <- burger]
```

We can also replace the comprehension expression in Python using `map`:

```python
def burger_price(burger):
    return sum(map(ingredient_price, burger))
```

Haskell also has a `map` (or `fmap`) function that does the same thing:

```haskell
burgerPrice burger = sum $ map ingredientPrice burger
```

The `$` sign is just regular function application, except that `$` binds very weakly. So `sum $ map ingredientPrice burger` is basically `sum (map ingredientPrice burger)`.

Finally, notice that `burgerPrice x = sum ((map ingredientPrice) x)`, so effectively we can finally define our function this way:

```haskell
burgerPrice = sum . map ingredientPrice
  where ingredientPrice i
          | i == 'B' = 0.5
          | i == 'C' = 0.8
          | i == 'P' = 1.5
          | i == 'V' = 0.7
          | i == 'O' = 0.4
          | i == 'M' = 0.9
```

To see this, let b be `burgerPrice`, g be `sum` and f be `map ingredientPrice`. We have shown that

$$b(x) = g(f(x))$$

By definition,

$$b = g \circ f$$

This style of writing functions is known as *point-free* style, where functions are expressed as a *composition* of functions.

Question 6

Again, there are several ways to solve this. To do so numerically, we can define our function recursively:

$$s(n) = \begin{cases} n & \text{if } n < 10 \\ n \mod 10 + s(\lfloor n \div 10 \rfloor) & \text{otherwise} \end{cases}$$

```haskell
sumDigits n
  | n < 10    = n
  | otherwise = n `mod` 10 + sumDigits (n `div` 10)
```

Alternatively, we may convert `n` into a string, convert each character into integers, then obtain the sum. This might be expressed in Python as:

```python
def sum_digits(n):
    return sum(map(int, str(n)))
```

Converting `n` into a string can be done by `show`:

```
ghci> show 123
"123"
```

Converting back into an integer can be done with `read` (you have to explicitly state the output type of the `read` function since this can be ambiguous):

```
ghci> read "123" :: Int
123
```

However, we can't `read` from **characters** since the `read` function receives strings. Good thing that strings are lists of characters, so by putting the character in a list, we now obtain the ability to read a digit (as a character) as an integer.

```
ghci> read '1' :: Int
-- error!
ghci> read ['1'] :: Int
1
```

To put things into lists, we can use the `return` function!

```
ghci> return '1' :: String
"1"
ghci> (read . return) '1' :: Int
1
```

Thus, the `read . return` function allows us to parse each character into an integer. Combining this with what we had before, we can obtain the list of the digits (as integers) from `n` using:

```
ghci> [(read . return) digit | digit <- show 123] :: [Int]
[1, 2, 3]
```

Again, we can use `map` instead of list comprehension.

```
ghci> map (read . return) (show 123) :: [Int]
[1, 2, 3]
```

Obtaining the sum of this list gives us exactly what we want. Thus, our `sumDigits` function is succinctly defined as follows:

```
sumDigits = sum . map (read . return) . show
```

Question 7

Idea: drop the first `start` elements, then take the `stop - start` elements after that.

```
    ls @: (start, stop) = take (stop - start) (drop start ls)
```

LAST UPDATED 26 OCT 2024

Types

Question 1

1. `Int`.
2. `String`. `x` has type `Int`, so `show x` has type `String`.
3. `String`. Recall that `String` is an alias for `[Char]`. Although the expression evaluates to `[]` which has type `forall a. [a]`, because both branches of the conditional expression must have the same type, the type of the expression is thus specialized into `[Char]`.
4. `[a] -> [a]`. `(++)` has type `forall a. [a] -> [a] -> [a]`, since `[]` is also polymorphic with type `forall a. [a]`, there is no need to specialize the resulting function call expression. This makes sense because any list can be concatenated with the empty list.
5. `[Int] -> [Int]`. The `map` function has type `(a -> b) -> [a] -> [b]`. Since we have supplied a function `Int -> Int`, we are thus specializing `a` and `b` to `Int`.
6. `(a -> [Int]) -> a -> String`. Recall that `(.)` has type `forall b c a. (b -> c) -> (a -> b) -> a -> c`. The function `\(x :: Int) -> show x` has type `Int -> String`. Thus, substituting `b` and `c` for `Int` and `String` respectively, we get our answer.
7. `(String -> a) -> Int -> a`. Note that `(+3)` is `\x -> x + 3`, while `(3+)` is `\x -> 3 + x`. As such, the answer here follows the same reasoning except that the argument to `(.)` is at the second position.
8. `(a, b) -> c -> (a, c)`. Note that `(,)` is the tuple (pair) constructor which has type `forall a, b. a -> b -> (a, b)`.
9. `(a -> Bool) -> [a] -> [a]`. As we know, `filter` receives a function that tests each element, and returns the list with only the elements that pass the test.

Question 2

1. `eqLast : Eq a => [a] -> [a] -> Bool`. This function can be polymorphic but requires that `a` is amenable to equality comparisons, so we add the `Eq` constraint to it. We will discuss more on typeclasses next week.
2. `isPalindrome : Eq a => [a] -> [a] -> Bool`. The reason for the `Eq` constraint is because we need to compare the two lists for equality, which means that the elements of both lists must be amenable to equality comparisons!

3. `burgerPrice : Fractional a => String -> a`. Notice once again that we have another typeclass constraint in this function signature. Typeclasses are incredibly common, and hopefully this might motivate you to understand these in the subsequent lectures. Nonetheless, if you had answered `String -> Double`, that is fair as well.
4. `@: : [a] -> (Int, Int) -> [a]`. The function receives a list, a pair of two integers, and produces a slice of the list of the same type.

Question 3

Let us first define a type that describes valid ingredients and a function on this type that gives their prices:

```
data Ingredient = B | C | P | V | O | M
price :: Ingredient -> Rational
price B = 0.5
price C = 0.8
price P = 1.5
price V = 0.7
price O = 0.4
price M = 0.9
```

Then, we can define a valid burger being a list of ingredients. For this, we can define a *type alias* like so:

```
type Burger = [Ingredient]
```

Type aliases are nothing special; more or less, they are *nicknames* for types. There is no difference between the `Burger` and `[Ingredient]` types, just like how there is no difference between `String` and `[Char]`. Then, we can define our `burgerPrice` function with pattern matching in a very standard way:

```
burgerPrice :: Burger -> Rational
burgerPrice [] = 0
burgerPrice (i : is) = price i + burgerPrice is
```

Let us take this a step further by observing the following function in Haskell's prelude:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f n [] = n
foldr f n (x : xs) =
  let r = foldr f n xs
  in  f x r
```

In practice, this does something very familiar:

$$\text{foldr}(f, n, [a_1, \ldots, a_n]) = f(a_1, f(a_2, \ldots f(a_{n-1}, f(a_n, n)) \ldots))$$

This looks like the right-associative equivalent of `reduce` in Python! (The equivalent of `reduce` in Haskell is the `foldl` function).

$$\text{reduce}(f, n, [a_1, \ldots, a_n]) = f(f(\ldots f(n, a_1), a_2), \ldots, a_n)$$

This hints to us that in the definition of `foldr`, `f` is the combiner function and `n` is the initial value. This corresponds very nicely to `burgerPrice`. Let us try rewriting our `burgerPrice` function to see this:

```
burgerPrice [] = 0
burgerPrice (x : xs) =
  let r = burgerPrice xs
      f a b = price a + b
      -- alternatively,
      -- f = (+) . price
  in  f x r
```

As you can see, if we let `f` be `(+) . price` and `n` be `0`, we can define `burgerPrice` based on `foldr`:

```
burgerPrice = foldr ((+) . price) 0
```

Question 4

Solutions are self-explanatory.

```
dropConsecutiveDuplicates :: Eq a => [a] -> [a]
dropConsecutiveDuplicates [] = []
dropConsecutiveDuplicates [x] = [x]
dropConsecutiveDuplicates (x : xx : xs)
  | x == xx   = dropConsecutiveDuplicates (x : xs)
  | otherwise = x : dropConsecutiveDuplicates (xx : xs)
```

Question 5

As hinted by the example runs, a zipper is a tuple of two lists. The idea is to model a zipper as two stacks. This is great because singly-linked lists (with head pointers), as we know, can model stacks.

```
type ListZipper a = ([a], [a])
mkZipper :: [a] -> ListZipper a
mkZipper ls = ([], ls)
```

Functions for traversing and replacing the elements of the zipper should be straightforward to define. Note that the `@` symbol binds the entire pattern on the right to the name on the left.

```
l, r :: ListZipper a -> ListZipper a

l x@([], _) = x
l (x : xs, ys) = (xs, x : ys)

r x@(_,[]) = x
r (xs, y : ys) = (y : xs, ys)

setElement :: a -> ListZipper a -> ListZipper a
setElement x (xs,[]) = (xs, [x])
setElement x (xs, _ : ys) = (xs, x : ys)
```

Question 6

To start, we define a binary tree. This is very similar to the tree examples that we have given, except that we allow the tree to be empty. Note that you might be tempted to put the `Ord` constraint at the data type declaration itself. This is deprecated, and also not recommended.

```
data SortedSet a = Empty | Node (SortedSet a) a (SortedSet a)
```

Let us start with the function to add elements to the sorted set. This should be straightforward if you remember how BST algorithms are defined.

```
(@+) :: Ord a => SortedSet a -> a -> SortedSet a
Empty @+ x = Node Empty x Empty
t@(Node left a right) @+ x
    | x == a     = t
    | x < a      = Node (left @+ x) a right
    | otherwise  = Node left a (right @+ x)
```

Given a BST, to get the list of elements in sorted order, perform an inorder traversal.

```
setToList :: SortedSet a -> [a]
setToList Empty = []
setToList (Node left a right) = setToList left ++ (a : setToList right)
```

Converting a list into a sorted set can be done by repeated applications of `@+` over the elements of the list. This should hint to us that we can use a fold over the list. Note that the `flip` function flips the arguments of a function: i.e. `flip f x y = f y x`.

```
sortedSet :: Ord a => [a] -> SortedSet a
sortedSet = foldr (flip (@+)) Empty
```

Finally, determining if an element is a member of the sorted set is a matter of binary search.

```haskell
in' :: Ord a => a -> SortedSet a -> Bool
in' _ Empty = False
in' x (Node left a right)
  | x == a    = True
  | x < a     = in' x left
  | otherwise = in' x right
```

An alternative to this implementation is to use AVL trees instead of plain BSTs. We provide an implementation of AVL trees at the end of this chapter.

Question 7

We start with the base definition which should be self-explanatory.

```haskell
-- Haskell
data Shape = Circle Double | Rectangle Double Double

area :: Shape -> Double
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass
from math import pi

class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        pass

@dataclass
class Circle(Shape):
    radius: float
    def area(self) -> float:
        return pi * self.radius ** 2

@dataclass
class Rectangle(Shape):
    width: float
    height: float
    def area(self) -> float:
        return self.width * self.height
```

We start with the first extension of our problem by creating a new shape called `Triangle`. Notice that to add representations of our types in our Haskell implementation, we must have access to edit whatever we've written before. This is unlike our OO implementation in

Python, where by adding a new shape, we can just define a completely separate subclass and define the `area` method for that class.

```haskell
data Shape = Circle Double
           | Rectangle Double Double
           | Triangle Double Double

area :: Shape -> Double
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
area (Triangle w h) = w * h / 2
```

```python
@dataclass
class Triangle(Shape):
    width: float
    height: float
    def area(self) -> float:
        return self.width * self.height / 2
```

However, proceeding with the second extension, we see that the opposite is true: adding a new function does not require edit access in our Haskell implementation since we can just define a separate function, but it is required for our Python implementation since we have to add this method to all the classes we have defined!

```haskell
scale :: Double -> Shape -> Shape
scale n (Circle r) = Circle (r * n)
scale n (Rectangle w h) = Rectangle (w * n) (h * n)
scale n (Triangle w h) = Triangle (w * n) (h * n)
```

```python
class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        pass
    @abstractmethod
    def scale(self, n: float) -> 'Shape':
        pass

@dataclass
class Circle(Shape):
    radius: float
    def area(self) -> float:
        return pi * self.radius ** 2
    def scale(self, n: float) -> Shape:
        return Circle(n * self.radius)

@dataclass
class Rectangle(Shape):
    width: float
    height: float
    def area(self) -> float:
        return self.width * self.height
    def scale(self, n: float) -> Shape:
        return Rectangle(self.width * n, self.height * n)

@dataclass
class Triangle(Shape):
    width: float
    height: float
    def area(self) -> float:
        return self.width * self.height / 2
    def scale(self, n: float) -> Shape:
        return Triangle(self.width * n, self.height * n)
```

Question 8

Defining additional constructors for our expressions GADT is relatively straightforward, and so is extending our `eval` function. We write the entire implementation here.

```haskell
{-# LANGUAGE GADTs #-}
data Expr α where
  LitNumExpr  :: Int -> Expr Int
  AddExpr     :: Expr Int -> Expr Int -> Expr Int
  EqExpr      :: Eq α => Expr α -> Expr α -> Expr Bool
  CondExpr    :: Expr Bool -> Expr α -> Expr α -> Expr α
  LitBoolExpr :: Bool -> Expr Bool
  AndExpr     :: Expr Bool -> Expr Bool -> Expr Bool
  OrExpr      :: Expr Bool -> Expr Bool -> Expr Bool
  FuncExpr    :: (α -> β) -> Expr (α -> β)
  FuncCall    :: Expr (α -> β) -> Expr α -> Expr β

eval :: Expr α -> α
eval (LitNumExpr n)   = n
eval (AddExpr a b)    = eval a + eval b
eval (EqExpr a b)     = eval a == eval b
eval (CondExpr a b c) = if eval a then eval b else eval c
eval (LitBoolExpr b)  = b
eval (AndExpr a b)    = eval a && eval b
eval (OrExpr a b)     = eval a || eval b
eval (FuncExpr f)     = f
eval (FuncCall f x)   = (eval f) (eval x)
```

Question 9

Bank Accounts

Bank Account ADT

As in the lecture notes, simulating ADTs in Python can be done either with an (abstract) class, or a type alias. In our case, we shall use the latter.

First, we create the type:

```haskell
type BankAccount = NormalAccount | MinimalAccount
```

Then, we create the `NormalAccount` and `MinimalAccount` classes:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class NormalAccount:
    account_id: str
    balance: float
    interest_rate: float

@dataclass(frozen=True)
class MinimalAccount:
    account_id: str
    balance: float
    interest_rate: float
```

Basic Features

For our two basic features, we shall employ a simple helper function that sets the amount of a bank account. Notice once again that we do not mutate any data structure in our program!

```
def _set_balance(amt: float, b: BankAccount) -> BankAccount:
    match b:
        case NormalAccount(id, _, i):
            return NormalAccount(id, amt, i)
        case MinimalAccount(id, _, i):
            return MinimalAccount(id, amt, i)
```

Then, the basic features can be defined in terms of our `_set_balance` helper function.

```
def deposit(amt: float, b: BankAccount) -> BankAccount:
    return _set_balance(b.balance + amt, b)

def deduct(amt: float, b: BankAccount) -> tuple[bool, BankAccount]:
    if amt > b.balance:
        return (False, b)
    return (True, _set_balance(b.balance - amt, b))
```

Advanced Features

At this point, implementing the advanced features should not be too difficult.

```
def _cmpd(p: float, r: float) -> float:
    return p * (1 + r)

def compound(b: BankAccount) -> BankAccount:
    match b:
        case NormalAccount(id, bal, i):
            return NormalAccount(id, _cmpd(bal, i), i)
        case MinimalAccount(id, bal, i):
            new_bal: float = max(bal - 20, 0) if bal < 1000 else bal
            return MinimalAccount(id, _cmpd(new_bal, i), i)

def transfer(amt: float, from_: BankAccount, to: BankAccount) -> tuple[bool,
BankAccount, BankAccount]:
    success: bool
    from_deducted: BankAccount
    success, from_deducted = deduct(amt, from_)
    if not success:
        return (False, from_, to)
    return (True, from_deducted, deposit(amt, to))
```

Operating on Bank Accounts

Operations ADT

The ADT definition is pretty straightforward:

```
type Op = Transfer | Compound

@dataclass
class Transfer:
    amount: float
    from_: str
    to: str

@dataclass
class Compound:
    pass
```

Processing One Operation

It's easier to write the functions that perform each individual operation first, especially since they are more involved with dictionary lookups etc. Take note of the fact that all of the data structures are unchanged!

```
# Type alias for convenience
type BankAccounts = dict[str, BankAccount]

def _compound_all(mp: BankAccounts) -> BankAccounts:
    return {k : compound(v) for k, v in mp.items()}

def _transfer(amt: float, from_: str, to: str, mp: BankAccounts) -> tuple[bool,
BankAccounts]:
    if from_ not in mp or to not in mp:
        return (False, mp)
    success: bool
    new_from: BankAccount
    new_to: BankAccount
    success, new_from, new_to = transfer(amt, mp[from_], mp[to])
    if not success:
        return (False, mp)
    new_mp: BankAccounts = mp | { from_: new_from, to: new_to }
    return (True, new_mp)
```

Then, the `process_one` function is easy to define since we can just invoke our helper functions:

```
def process_one(op: Op, mp: BankAccounts) -> tuple[bool, BankAccounts]:
    match op:
        case Transfer(amt, from_, to):
            return _transfer(amt, from_, to, mp)
        case Compound():
            return (True, _compound_all(mp))
```

Process All Operations

Given the `process_one` function, the `process_all` function should be straightforward. Note once again that none of the data structures are being mutated and we use recursion. The last `case` statement is only used to suppress `pyright` warnings.

```
def process_all(ops: list[Op], mp: BankAccounts) -> tuple[list[bool],
BankAccounts]:
    match ops:
        case []:
            return [], mp
        case x, *xs:
            op_r, mp1 = process_one(x, mp)
            rs, mp2 = process_all(xs, mp1)
            return [op_r] + rs, mp2
        case _: raise
```

Polymorphic Processing

Notice that if we had received the `process_one` function as an argument then we would now have a higher-order function:

```
from typing import Callable
# For brevity
type P = Callable[[Op, BankAccounts], tuple[bool, BankAccounts]]
def process_all(process_one: P, ops: list[Op], mp: BankAccounts) ->
tuple[list[bool], BankAccounts]:
    match ops:
        case []:
            return [], mp
        case x, *xs:
            op_r, mp1 = process_one(x, mp)
            rs, mp2 = process_all(process_one, xs, mp1)
            return [op_r] + rs, mp2
        case _: raise
```

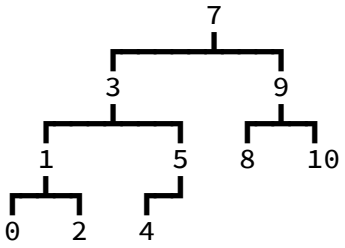
Now notice that `process_all`'s implementation does not depend on `Op`, `bool` or `BankAccounts`. Let us make this function polymorphic by replacing `Op` with `A`, `BankAccounts` with `B` and `bool` with `C`!

```
def process[A, B, C](f: Callable[[A, B], tuple[C, B]], ops: list[A], mp: B) ->
tuple[list[C], B]:
    match ops:
        case []:
            return [], mp
        case x, *xs:
            op_r, mp1 = f(x, mp)
            rs, mp2 = process(f, xs, mp1)
            return [op_r] + rs, mp2
        case _: raise
```

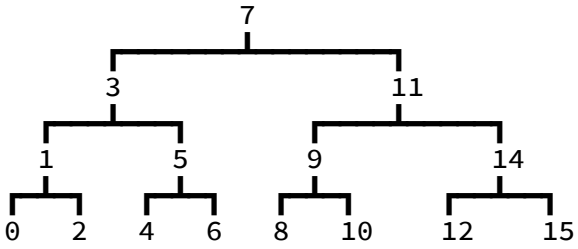
AVL Trees

Here we show an example of using AVL trees as sorted sets. Notice our AVL tree has nice pretty printing, pretty cool huh! We will learn how to define the string representation of a type in subsequent lectures.

```
ghci> x = fromList [1,1,1,2,2,2,8,5,4,3,5,9,0,10,0,7,8,3]
ghci> x
```




```
ghci> x @+ 6 @+ 11 @+ 14 @+ 12 @+ 15
```



We first start with some declarations and imports.

```haskell
module Avl ( AVL(Empty), in', toList, fromList, (@+)) where

import Data.List (intercalate)

data AVL a = Empty | Node (AVL a) a (AVL a)
  deriving Eq

in'      :: Ord a => a -> AVL a -> Bool
toList   :: AVL a -> [a]
fromList :: Ord a => [a] -> AVL a
(@+)     :: Ord a => AVL a -> a -> AVL a
infixl 7 @+
```

Next, we provide implementations of these declarations. Many of these are identical to that of our sorted set implementation using BSTs; the only difference is in `@+` where AVL trees have to perform height balancing if the balance factor exceeds the range $[-1, 1]$.

```
in' _ Empty = False
in' x (Node left a right)
  | x == a    = True
  | x < a     = in' x left
  | otherwise = in' x right

toList Empty = []
toList (Node left a right) = toList left ++ (a : toList right)

fromList = foldr (flip (@+)) Empty

Empty @+ x = Node Empty x Empty
o@(Node left a right) @+ x
  | x < a =
      let newLeft = left @+ x
          newTree = Node newLeft a right
      in  if bf newTree > -2 then newTree
          else
            let t
                  | bf newLeft > 0 = Node (rotateLeft newLeft) a right
                  | otherwise      = newTree
            in rotateRight t
  | x > a =
      let newRight = right @+ x
          newTree = Node left a newRight
      in  if bf newTree < 2 then newTree
          else let t
                     | bf newRight < 0 = Node left a (rotateRight newRight)
                     | otherwise       = newTree
               in rotateLeft t
  | otherwise = o
```

The implementation of these functions involve some additional helper functions for obtaining balance factors and rotations, which we declare and define here:

```
-- Implementation helpers
height :: AVL a -> Int
height Empty = 0
height (Node left _ right) = 1 + max (height left) (height right)

rotateLeft :: AVL a -> AVL a
rotateLeft Empty = Empty
rotateLeft t@(Node _ _ Empty) = t
rotateLeft (Node left a (Node ll b right)) = Node (Node left a ll) b right

rotateRight :: AVL a -> AVL a
rotateRight Empty = Empty
rotateRight t@(Node Empty _ _) = t
rotateRight (Node (Node left b rr) a right) = Node left b (Node rr a right)

bf :: AVL a -> Int -- balance factor
bf Empty = 0
bf (Node l _ r) = height r - height l
```

Finally, we write functions to support pretty printing.

```haskell
-- Pretty printing
strWidth :: Show a => AVL a -> Int
strWidth Empty = 0
strWidth (Node left a right) =
  let leftWidth = strWidth left
      l = if leftWidth > 0 then leftWidth + 1 else 0
      centerWidth = length $ show a
      rightWidth = strWidth right
      r = if rightWidth > 0 then rightWidth + 1 else 0
  in  l + centerWidth + r

leftPad :: Int -> String -> String
leftPad 0 s = s
leftPad n s = leftPad (n - 1) (' ' : s)

rightArm, leftArm :: Int -> String

rightArm n = aux n where
  aux n'
    | n' == n   = '┗' : aux (n' - 1)
    | n' > 0    = '━' : aux (n' - 1)
    | otherwise = "┓"

leftArm n = aux n where
  aux n'
    | n' == n = '┏' : aux (n' - 1)
    | n' > 0  = '━' : aux (n' - 1)
    | otherwise = "┛"

bothArm :: Int -> Int -> String
bothArm mid right = aux 0 where
  aux n'
    | n' == 0 = '┏' : aux 1
    | n' /= mid && n' < right = '━' : aux (n' + 1)
    | n' == mid = '┻' : aux (n' + 1)
    | otherwise = "┓"

toRowList :: Show a => AVL a -> [String]
toRowList Empty = []
toRowList (Node Empty a Empty) = [show a]
toRowList (Node Empty a right) =
  let x = toRowList right
      nodeLength = length $ show a
      y = map (leftPad (nodeLength + 1)) x
      rroot = rootAt right + nodeLength + 1
  in show a : rightArm rroot : y
toRowList (Node left a Empty) =
  let x = toRowList left
      lroot = rootAt left
      nodeAt = strWidth left + 1
  in leftPad nodeAt (show a) : leftPad lroot (leftArm (nodeAt - lroot)) : x
toRowList (Node left a right) =
  let l = toRowList left
      r = toRowList right
      lw = strWidth left
      rpadding = lw + 2 + length (show a)
      rr = zipStringTree rpadding l r
```

```
        lroot = rootAt left
        rroot = rootAt right
        nodeAt = lw + 1
        f = leftPad (lw + 1) (show a)
        s = leftPad lroot (bothArm (nodeAt - lroot) (rroot - lroot + rpadding))
    in  f : s : rr


rightPadTo :: Int -> String -> String
rightPadTo n s
  | ls >= n   = s
  | otherwise = let n' = n - ls
                    s' = leftPad n' []
                in  s ++ s'
  where ls = length s

rootAt :: Show a => AVL a -> Int
rootAt Empty = 0
rootAt (Node Empty _ _) = 0
rootAt (Node left _ _) = strWidth left + 1

zipStringTree :: Int -> [String] -> [String] -> [String]
zipStringTree _ [] [] = []
zipStringTree _ l [] = l
zipStringTree n [] r = map (leftPad n) r
zipStringTree n (l : ls) (r : rs) =
  let res = zipStringTree n ls rs
      c   = rightPadTo n l ++ r
  in  c : res

instance Show a => Show (AVL a) where
  show Empty = ""
  show t = intercalate "\n" $ toRowList t
```

LAST UPDATED 26 OCT 2024

Typeclasses

Question 1

1. `Num a => a`. Because all of `1`, `2` and `3` can be interpreted as any number, the entire expression can likewise be interpreted as any number.
2. `Show b => (a -> b) -> a -> String`. The type of `show` is `Show a => a -> String`, in other words, any type that implements the `Show` typeclass can be converted into a `String`. Therefore, `(show .)` can receive any function `a -> b` where `b` implements `Show`, so that the result is a function that receives `a` and produces `String`
3. `Show a => (String -> b) -> a -> b`. Similar to the above.
4. `Eq a => (a, a) -> Bool`. The elements of the tuple must be amenable to equality comparisons, and therefore must be of the same type `a` where `a` implements `Eq`.

Question 2

The idea is to create a protocol that describes classes that have a `to_list` function. In the following solution, the protocol is called `ToList`.

```python
from typing import Any

type Tree[a] = Empty | TreeNode[a]
type List[a] = Empty | ListNode[a]

@dataclass
class Empty:
    def to_list(self) -> list[Any]:
        return []

@dataclass
class ListNode[a]:
    head: a
    tail: List[a]
    def to_list(self) -> list[a]:
        return [self.head] + self.tail.to_list()

@dataclass
class TreeNode[a]:
    l: Tree[a]
    v: a
    r: Tree[a]
    def to_list(self) -> list[a]:
        return self.l.to_list() + [self.v] + self.r.to_list()

class ToList[a](Protocol):
    def to_list(self) -> list[a]:
        raise

def flatten[a](ls: list[ToList[a]]) -> list[a]:
    if not ls: return []
    return ls[0].to_list() + flatten(ls[1:])

ls: list[ToList[int]] = [ListNode(1, Empty()), TreeNode(Empty(), 2, Empty())]
ls2: list[int] = flatten(ls)
```

Question 3

The `smallest` function can be implemented directly with the `minBound` method of the `Bounded` typeclass:

```haskell
smallest :: Bounded a => a
smallest = minBound
```

The `descending` function can also be implemented directly with the `Bounded` and `Enum` methods. The idea is to construct a range (which requires `Enum`) starting from `maxBound` and enumerating all the way to `minBound`. You can either construct a range starting from `minBound` to `maxBound` and then reverse the list, or you can start from `maxBound`, followed by `pred maxBound` (`pred` comes from `Enum`), and end at `minBound`.

```haskell
descending :: (Bounded a, Enum a) => [a]
descending = [maxBound,pred maxBound..minBound]
```

The `average` function can be implemented by converting the two terms to integers using `fromEnum`, then take the average, and use `toEnum` to bring it back to the desired term.

```haskell
average :: Enum a => a -> a -> a
average x y = toEnum $ (fromEnum x + fromEnum y) `div` 2
```

Question 4

Any list of elements that can be ordered, i.e. any list over a type implementing `Ord` can be sorted!

```haskell
import Data.List (splitAt)
mergesort :: Ord a => [a] -> [a]
mergesort ls
  | len <= 1 = ls
  | otherwise = let (l, r) = splitAt (len `div` 2) ls
                    l'     = mergesort l
                    r'     = mergesort r
                in  merge l' r'
  where len :: Int
        len = length ls
        merge :: Ord a => [a] -> [a] -> [a]
        merge [] x = x
        merge x [] = x
        merge l@(x : xs) r@(y : ys)
          | x <= y = x : merge xs r
          | otherwise = y : merge l ys
```

Question 5

Before we even begin, it will be helpful to decide what our typeclass will look like. The typeclass should be abstracted over the type of expression and the type from evaluating it. Therefore, it should be something like `Expr e a`, where `eval :: e -> a`. However, we know that `e` uniquely characterizes `a`, therefore we should add this as a functional dependency of our typeclass.

```haskell
class Expr e a | e -> a
  eval :: e -> a

-- for clarity
type IntExpr e = Expr e Int
type BoolExpr e = Expr e Bool
```

Then, our types will all contain types that implement the `Expr` typeclass.

First, to start we have numeric literals, which is straightforward.

```
data LitNumExpr = LitNumExpr Int

instance Expr LitNumExpr Int where
  eval :: LitNumExpr -> Int
  eval (LitNumExpr x) = x
```

`AddExpr` is more interesting. We require that the component expressions must be evaluated to an `Int`. As such, we constrain the component addends with `IntExpr` as follows:

```
data AddExpr where
  AddExpr :: (IntExpr e, IntExpr e') => e -> e' -> AddExpr

instance Expr AddExpr Int where
  eval :: AddExpr -> Int
  eval (AddExpr e1 e2) = eval e1 + eval e2
```

To define `EqExpr`, we have to allow expressions of any type that evaluates to any type that is amenable to equality comparisons:

```
data EqExpr where
  EqExpr :: (Eq a, Expr e a, Expr e' a) => e -> e' -> EqExpr

instance Expr EqExpr Bool where
  eval :: EqExpr -> Bool
  eval (EqExpr e1 e2) = eval e1 == eval e2
```

Finally, to define a `CondExpr` we must allow it to evaluate to any type, and thus should be parameterized.

```
data CondExpr a where
  CondExpr :: (BoolExpr c, Expr e a, Expr e' a)
           => c -> e -> e' -> CondExpr a

instance Expr (CondExpr a) a where
  eval :: CondExpr a -> a
  eval (CondExpr c e1 e2) = if eval c then eval e1 else eval e2
```

Question 6

As per usual, we are going to define a typeclass `Sequence` that defines the methods `@`, `len` and `prepend`. The type parameters of `Sequence` is tricky. One possibility is for `Sequence` to be higher-kinded:

```haskell
class Sequence e s where
    (@) :: s e -> Int -> e
    len :: s e -> Int
    prepend :: s e -> e -> s e

instance Sequence [] a where
    -- ...
```

However, this will not work when having `Int`s as sequences because `Int` is not a type constructor. Therefore, we will just let `s` be the full sequence type, and introduce a functional dependency `s -> e` so that the sequence type `s` *uniquely characterizes* the type of the elements of that sequence:

```haskell
class Sequence e s | s -> e where
    (@) :: s -> Int -> e
    len :: s -> Int
    prepend :: s -> e -> s
```

In which case, the `Sequence` instances for `[a]` and `Int` becomes quite straightforward:

```haskell
instance Sequence a [a] where
  (@) :: [a] -> Int -> a
  (@) = (!!)

  len :: [a] -> Int
  len = length

  prepend :: [a] -> a -> [a]
  prepend = flip (:)

instance Sequence () Int where
  (@) :: Int -> Int -> ()
  i @ j
    | j < 0 || j >= i = undefined
    | otherwise       = ()

  len :: Int -> Int
  len = id

  prepend :: Int -> () -> Int
  prepend = const . (+1)
```

LAST UPDATED **26 OCT 2024**

Question 1

To implement these classes and methods, just "convert" the Haskell definitions to Python code. Note that `Validation` is *not* a monad.

```python
from typing import Any
from dataclasses import dataclass

class List:
    @staticmethod
    def pure(x): return Node(x, Empty())

    # Convenience method for Question 3
    @staticmethod
    def from_list(ls):
        match ls:
            case []: return Empty()
            case x, *xs: return Node(x, List.from_list(xs))

@dataclass
class Node(List):
    head: object
    tail: List

    def map(self, f):
        return Node(f(self.head), self.tail.map(f))

    def ap(self, x):
        tails = self.tail.ap(x)
        heads = Node._ap(self.head, x)
        return heads.concat(tails)

    # helper method
    @staticmethod
    def _ap(f, xs):
        match xs:
            case Empty(): return Empty()
            case Node(l, r): return Node(f(l), Node._ap(f, r))

    def concat(self, xs):
        return Node(self.head, self.tail.concat(xs))

    def flatMap(self, f):
        return f(self.head).concat(self.tail.flatMap(f))

@dataclass
class Empty(List):
    def map(self, f): return self
    def concat(self, xs): return xs
    def ap(self, x): return self
    def flatMap(self, f): return self

class Maybe:
    @staticmethod
    def pure(x): return Just(x)

@dataclass
class Just(Maybe):
    val: object

    def map(self, f): return Just(f(self.val))
```

```
        def ap(self, x):
            match x:
                case Just(y): return Just(self.val(y))
                case Nothing(): return x

        def flatMap(self, f): return f(self.val)

    @dataclass
    class Nothing:
        def map(self, f): return self
        def ap(self, x): return self
        def flatMap(self, f): return self

    class Either:
        @staticmethod
        def pure(x): return Right(x)

    @dataclass
    class Left(Either):
        inl: object
        def map(self, f): return self
        def ap(self, f): return self
        def flatMap(self, f): return self

    @dataclass
    class Right(Either):
        inr: object

        def map(self, f): return Right(f(self.inr))

        def ap(self, f):
            match f:
                case Left(e): return f
                case Right(x): return Right(self.inr(x))

        def flatMap(self, f): return f(self.inr)

    class Validation:
        @staticmethod
        def pure(x): return Success(x)

    @dataclass
    class Success:
        val: object

        def map(self, f): return Success(f(self.val))

        def ap(self, f):
            match f:
                case Failure(e): return f
                case Success(x): return Success(self.val(x))


    @dataclass
    class Failure:
        err: list[str]

        def map(self, f): return self
```

```
    def ap(self, f):
        match f:
            case Failure(err): return Failure(self.err + err)
            case Success(x): return self
```

Question 2

Question 2.1: Unsafe Sum

The Python implementation of `sum_digits` can be a Haskell rewrite of your `sumDigits` solution for Question 6 in Chapter 1.4 (Course Introduction#Exercises):

```
def sum_digits(n):
    return n if n < 10 else \
           n % 10 + sum_digits(n // 10)
```

Question 2.2: Safe Sum

The idea is to have `sum_digits` return a `Maybe` object. In particular, the function should return `Nothing` if `n` is negative, and `Just x` when `n` is positive and produces result `x`.

```
def sum_digits(n):
    return Nothing() if n < 0 else \
           Just(n)   if n < 10 else \
           sum_digits(n // 10).map(lambda x: x + n % 10)
```

```
sumDigits :: Int -> Maybe Int
sumDigits n
  | n < 0 = Nothing
  | n < 10 = Just n
  | otherwise = (n `mod` 10 +) <$> sumDigits (n `div` 10)
```

Question 2.3: Final Sum

The result of `sum_digits` is a `Maybe[int]`, and `sum_digits` itself has type `int -> Maybe[int]`. To compose `sum_digits` with itself we can use `flatMap` or `>>=`.

```
def final_sum(n):
    n = sum_digits(n)
    return n.flatMap(lambda n2: n2 if n2 < 10 else final_sum(n2))
```

```haskell
finalSum :: Int -> Maybe Int
finalSum n = do
  n' <- sumDigits n
  if n' < 10
  then Just n'
  else finalSum n'
```

Question 3

Question 3.1: Splitting Strings

`split` in Python can be implemented with the `str.split` method. The `split` function for Haskell is shown in Chapter 4.4 (Railway Pattern#Validation).

```python
# Uses the convenience method from_list in the List class
def split(char, s):
    return List.from_list(s.split(char))
```

Question 3.2: CSV Parsing

Split the string over `\n`, then split each string in that list over `,` using `map`:

```python
def csv(s):
    return split('\n', s)
                .map(lambda x: split(',', x))
```

```haskell
csv :: String -> [[String]]
csv s = split ',' <$> (split '\n' s)
```

Question 4

Question 4.1: Factorial

Should be boring at this point.

```python
def factorial(n):
    return 1 if n <= 1 else \
           n * factorial(n - 1)
```

```haskell
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Question 4.2: Safe Factorial

The idea is to return a `Left` if `n` is negative, `Right` with the desired result otherwise. Typically, `Right` is the happy path.

```python
def factorial(n, name):
    if n < 0:
        return Left(name + ' cannot be negative!')
    if n <= 1:
        return Right(1)
    return factorial(n - 1, name).map(lambda x: x * n)
```

```haskell
factorial :: Int -> String -> Either String Int
factorial n name
  | n < 0 = Left $ name ++ " cannot be negative!"
  | n <= 1 = Right 1
  | otherwise = (n*) <$> factorial (n - 1) name
```

Question 4.3: Safe n choose k

Idea: Compute $\mathrm{n}!$, $\mathrm{k}!$ and $(\mathrm{n}-\mathrm{k})!$ in "parallel", combine with `ap`:

```python
def choose(n, k):
    nf = factorial(n, 'n')
    kf = factorial(k, 'k')
    nmkf = factorial(n - k, 'n - k')
    div = lambda x: lambda y: lambda z: x // y // z
    return nf.map(div).ap(kf).ap(nmkf)
```

```haskell
choose :: Int -> Int -> Either String Int
choose n k
    = let nf   = factorial n "n"
          kf   = factorial k "k"
          nmkf = factorial (n - k) "n - k"
          f x y z = x `div` y `div` z
      in f <$> nf <*> kf <*> nmkf
```

With the `ApplicativeDo` language extension enabled, you can just use `do` notation:

```haskell
{-# LANGUAGE ApplicativeDo #-}
choose :: Int -> Int -> Either String Int
choose n k = do
  nf <- factorial n "n"
  kf <- factorial k "k"
  nmkf <- factorial (n - k) "n - k"
  return $ nf `div` kf `div` nmkf
```

Question 4.4

Redefine `factorial` to use `Validation` instead of `Either`:

```python
def factorial(n, name):
    if n < 0:
        return Failure([f'{name} cannot be negative!'])
    if n <= 1:
        return Success(1)
    else:
        return factorial(n - 1, name).map(lambda x: n * x)
```

```haskell
factorial :: Int -> String -> Validation [String] Int
factorial n name
  | n < 0 = Failure [name ++ " cannot be negative!"]
  | n <= 1 = Success 1
  | otherwise = (n*) <$> factorial (n - 1) name
```

Finally, update the type signature of `choose` (we do not need to do so in Python).

```haskell
choose :: Int -> Int -> Validation [String] Int
choose n k = do
  nf <- factorial n "n"
  kf <- factorial k "k"
  nmkf <- factorial (n - k) "n - k"
  return $ nf `div` kf `div` nmkf
```

LAST UPDATED **26 OCT 2024**

In this chapter we will do a brief recap of some of the basic concepts you *might* have learnt in IT5001. If you haven't, fret not. The recap should provide enough context for you to read the rest of these notes.

LAST UPDATED 26 OCT 2024

Recursion

Something is recursive if it is defined using itself. A simple (albeit hardly useful and contrived) example is the following function:

```
def f(n):
    return f(n + 1)
```

As defined, the body of function `f` invokes itself. In other words, it is *defined using itself*. Readers who are unconvinced that `f` is not a recursive definition may see that it is analogous to the following mathematical definition, which is clearly recursive:

$$\mathrm{f}(\mathrm{n}) = \mathrm{f}(\mathrm{n}+1) = \mathrm{f}(\mathrm{n}+2) = \mathrm{f}(\mathrm{n}+3) = \ldots$$

Data types can also be defined recursively:

```
from abc import ABC
from dataclasses import dataclass

class SinglyLinkedList(ABC):
    pass

class Empty(SinglyLinkedList):
    pass

@dataclass
class Node(SinglyLinkedList):
    head: object
    tail: SinglyLinkedList
```

Likewise, you can see that the `SinglyLinkedList` class has a subclass `Node` which itself holds another `SinglyLinkedList`. This makes `SinglyLinkedList` a recursive data structure.

The core idea we present in this section is that we can write recursive functions by thinking *structural-inductively*.

Induction

We shall begin by describing a *proof by induction* for a statement over the natural numbers. The principle of a proof by induction is as follows: given a *predicate* $\mathrm{P}(\mathrm{n})$ over the natural numbers, if we can show:

1. $P(0)$ is true
2. $\forall n \in \mathbb{N}.\ P(n) \rightarrow P(n+1)$ (for all natural numbers n, $P(n)$ implies $P(n+1)$)

Then $P(n)$ is true for all natural numbers n. This works because of *modus ponens*.

$$\frac{p \qquad p \rightarrow q}{q}\text{Modus Ponens}$$

Modus Ponens codifies the following idea: if a proposition p is true, and if p implies q, then q is true. To show how this allows proofs by induction, we see that we have a proof of $P(0)$. Since we also know that $P(0)$ implies $P(0+1) = P(1)$, by *modus ponens*, $P(1)$ is true. We also know that $P(1)$ implies $P(2)$, and since from earlier $P(1)$ is true, by *modus ponens*, $P(2)$ is also true, and so on.

$$\frac{P(0) \qquad \forall k \in \mathbb{N}.\, P(k) \rightarrow P(k+1)}{\forall n \in \mathbb{N}.\, P(n)}\text{Induction}$$

Let us attempt to write a proof by induction. We start with an implementation of the factorial function, then prove that it is correct:

```
def factorial(n):
    return 1 if not n else \
           n * factorial(n - 1)
```

> *Proposition*. Let $P(n)$ be the proposition that `factorial(n)` returns $n!$. Then, for all natural numbers n, $P(n)$ is true.

Proof. We prove $P(0)$ and $\forall n \in \mathbb{N}.\ P(n) \rightarrow P(n+1)$ separately.

Basis. Trivial. $0! = 1$. Furthermore, by definition, `factorial(0)` returns `1`. In other words, $P(0)$ is true.

Inductive. Suppose for some natural number k, `factorial(k)` returns $k! = k \times (k-1) \times \cdots \times 1$.

- By definition of `factorial`, `factorial(k + 1)` returns `(k + 1) * factorial(k)`.
- By our supposition, this evaluates to $(k+1) \times k!$, which is, by definition, $(k+1)!$.

Thus, if for some k, `factorial(k)` returns $k!$, then `factorial(k + 1)` returns $(k+1)!$. In other words, $\forall k \in \mathbb{N}.\ P(k) \rightarrow P(k+1)$.

As such, since we have proven $P(0)$ and $\forall k \in \mathbb{N}.\ P(k) \rightarrow P(k+1)$, we have proven $\forall n \in \mathbb{N}.\ P(n)$ by induction. □

Recursion via Inductive Reasoning

Naturally (haha), the next question to ask would be, "how do we make use of induction to write recursive functions?" As above, the recipe for a proof by induction involves (broadly) two steps:

1. Proof of the basis, e.g. $\mathrm{P}(0)$
2. The inductive proof, e.g. $\mathrm{P}(\mathrm{k}) \longrightarrow \mathrm{P}(\mathrm{k}+1)$. Typically, the inductive step is completed by **supposing** $\mathrm{P}(\mathrm{k})$ for some k, and showing $\mathrm{P}(\mathrm{k}+1)$.

We can write recursive functions similarly by providing:

1. Non-recursive computation for the result of the base-case, e.g. $\mathrm{f}(0)$;
2. Recursive computation of $\mathrm{f}(\mathrm{k}+1)$ based on the result of $\mathrm{f}(\mathrm{k})$ **assuming** that $\mathrm{f}(\mathrm{k})$ gives the correct result.

Let us start with a simple description of the natural numbers:

$$
\begin{array}{ll}
0 \in \mathbb{N} & \triangleright\ 0 \text{ is a natural number} \\
\mathrm{n} \in \mathbb{N} \longrightarrow \mathrm{S}(\mathrm{n}) \in \mathbb{N} & \triangleright\ \text{if n is a natural number then it has a successor that is also a}
\end{array}
$$

In our usual understanding of the natural numbers, $\mathrm{S}(\mathrm{n}) = \mathrm{n}+1$.

A formulation of the natural numbers in Python might be the following:

```
class Nat: pass

@dataclass
class Zero(Nat): pass

@dataclass
class Succ(Nat):
    pred: Nat
```

In which case, the number 3 can be written as follows:

```
three = Succ(Succ(Succ(Zero())))
```

Let us attempt to define addition over the natural numbers as we have formulated above, recursively:

```
>>> three = Succ(Succ(Succ(Zero())))
>>> two = Succ(Succ(Zero()))
>>> add(three, two)
Succ(pred=Succ(pred=Succ(pred=Succ(pred=Succ(pred=Zero())))))
```

We might decide to perform recursion on the first addend (doing so on the second addend is fine as well). In computing $\mathrm{m}+\mathrm{n}$ there are two possibilities for what m could be:

- 0, or
- the successor of some natural number k.

The first case is straightforward since 0 itself is non-recursive (see the definition of `Zero` above), and $0 + n$ is just n. In the other case of $m + n$ where $m = S(k) = k + 1$ for some k, assuming (via our inductive hypothesis) that `add(k, n)` correctly gives $k + n$, then $m + n$ is $(k + n) + 1$ which can be done by `Succ(add(k, n))`.

Therefore, we arrive at the following solution:

```
def add(m, n):
    return n if m == Zero() else \
           Succ(add(m.pred, n))
```

Using *structural pattern matching* which we present in Chapter 2.4 (Pattern Matching), we may also write the following definition which might be more intuitive:

```
def add(m, n):
    match m:
        case Zero(): return n
        case Succ(k): return Succ(add(k, n))
```

At this point you might be wondering why we had given such an odd formulation of the natural numbers in Python, when we could have used the `int` type instead (we totally could). One core idea we would like to make apparent in this formulation, is that recursion via inductive reasoning can be done over the *structure* of data. Our formulation shows that natural numbers are recursive data structures, where the successor of a natural number has a predecessor who is also, likewise, a natural number. This should make writing recursive functions over other kinds of recursive data structures not too great of a leap from writing recursive functions over natural numbers. To show this, consult our `SinglyLinkedList` data structure from above before we proceed to write recursive functions over them using inductive reasoning.

First, we shall write a function that appends an element to the end of a singly-linked list.

```
>>> append(1, Empty())
Node(head=1,tail=Empty())
>>> append(2, append(1, Empty()))
Node(head=1,tail=Node(head=2,tail=Empty()))
```

We can perform recursion over the structure of the list. There are two possible structures of the list:

1. The empty list
2. A node of a head element and a tail list

In the former, we append to an empty list, which should give the singleton. Note once again that because the empty list is non-recursive, our solution for appending to the empty list

likewise requires no recursion. For the second case of $[e_1, e_2, \ldots, e_n]$ (shorthand for $\text{Node}(e_1, [e_2, \ldots, e_n])$), assume that our solution is correct for the substructure of the `Node`, i.e. $\text{append}(x, [e_2, \ldots, e_n]) = [e_2, \ldots, e_n, x]$. Our goal is to have

$$\text{append}(x, \text{Node}(e_1, [e_2, \ldots, e_n])) = \text{Node}(e_1, [e_2, \ldots, e_n, x])$$

Observe that:

$$\begin{aligned}\text{append}(x, \text{Node}(e_1, [e_2, \ldots, e_n])) &= \text{Node}(e_1, [e_2, \ldots, e_n, x]) \\ &= \text{Node}(e_1, \text{append}(x, [e_2, \ldots, e_n]))\end{aligned}$$

Therefore, we can write:

```
def append(x, ls):
    if ls == Empty():
        return Node(x, Empty())
    return Node(ls.head, append(x, ls.tail))

# Using structural pattern matching:
def append2(x, ls):
    match ls:
        case Empty():
            return Node(x, Empty())
        case Node(e1, xs):
            return Node(e1, append2(x, xs))
```

We shall give another example by writing list reversals recursively, going straight into our derivation. Reversing the empty list gives the empty list. For nonempty lists our goal is to have $\text{reverse}([e_1, \ldots, e_n]) = [e_n, \ldots, e_1]$. Assuming that $\text{reverse}([e_2, \ldots, e_n]) = [e_n, \ldots, e_2]$, we can see that $[e_n, \ldots, e_1] = \text{append}(e_1, [e_n, \ldots, e_2])$, giving us the following formulation:

```
def reverse(ls):
    if ls == Empty():
        return Empty()
    return append(ls.head, reverse(ls.tail))

# Using structural pattern matching:
def reverse2(ls):
    match ls:
        case Empty(): return Empty()
        case Node(e1, xs): return append(e1, reverse2(xs))
```

By this point you should be able to see that recursion can be done via the following based on the structure of the data:

1. If the structure of the data is non-recursive, provide a non-recursive computation that computes the result directly
2. If the structure of the data is recursive, recursively solve the problem on the substructure(s) of the data (e.g. `pred` or `tail` of the natural number or list), and include its result in your main result

You should be well aware that data structures may be more complex. For example, solving a problem for a structure may require more than one recursive calls, one non-recursive call and one recursive call, etc. To make this apparent, let us look at a formulation of a binary tree of integers:

```
class Tree: pass

@dataclass
class EmptyTree(Tree): pass

@dataclass
class TreeNode(Tree):
    left: Tree
    val: int
    right: Tree
```

Now let us attempt to write a function that sums all integers in the tree. Again there are two possible structures a tree can have: the first being the empty tree, which has sum 0. For tree nodes, we have two subtrees, `left` and `right`, from whom we may recursively obtain their sums using our function. Then, the sum of the entire tree is just the total of the value at the node, the sum of the left subtree and the sum of the right subtree:

```
def sum_tree(t):
    if t == EmptyTree():
        return 0
    return t.val + sum_tree(t.left) + sum_tree(t.right)

# Structural pattern matching
def sum_tree(t):
    match t:
        case EmptyTree(): return 0
        case TreeNode(l, v, r):
            return sum_tree(l) + v + sum_tree(r)
```

In summary, our formulation of the natural numbers reveals that numbers are also structurally recursive, and therefore, are amenable to recursive computations. We can extend this idea to all recursive structures, which as you will see in these notes, is very common.

LAST UPDATED 26 OCT 2024

First-Class Functions

When we say that a language has first-class functions, what we mean is that functions are just regular terms or objects just like other terms and objects that you frequently encounter. Therefore, they can be *assigned to variables*, *passed in as arguments* and *returned from functions*. A language like Python (and of course, functional programming languages like Haskell and Lean) has first-class functions, making the following program completely valid:

```
def foo():
    return 1
x = foo
y = x() # 1
```

Although this program seems extremely weird, especially for those who are familiar with languages like C and Java, it totally works. The idea is, at least in Python, that functions are also *objects*, and therefore the `foo` *name* or *variable* actually stores a reference to the function that always returns `1`. This reference can be assigned to any other variable like `x` because `foo` is also a reference to an object! Then, when we invoke `x`, the Python runtime looks-up the reference stored in `x` which points to the `foo` function, and thus evaluates to `1`.

Then, a function that receives functions as arguments or returns functions is known as a *higher-order function*. Let us look at the following examples:

```
def add(x):
    def add_x(y):
        return x + y
    return add_x
```

Invoking this function is slightly weird, although still behaves more-or-less as expected:

```
>>> add(1)(2)
3
```

As you can see, `add` defines a local function `add_x` that receives `y` and returns `x + y`, for whatever `x` was passed into `add`. Then, `add` returns the `add_x` function itself! Therefore, `add(1)` actually evaluates to the *function* `add_x` where `x` is `1`, and when *that* is invoked, it evaluates to `1 + 2` which is `3`! This is an example of a function that *returns* a function, making it a higher-order function.

Another example is as follows:

```
def map(f, it):
    return (f(i) for i in it)
```

This function *invokes* the argument `f`, passing each `i` from `it`. Therefore, `f` is a function! An example of using `map` is as follows:

```
>>> def add_1(x): return x + 1
>>> list(map(add_1, [1, 2, 3]))
[2, 3, 4]
```

As you can see, `map` applies `add_1` to every single element of `[1, 2, 3]` and yields them into the resulting list, thereby giving us `[2, 3, 4]`! Again, since `map` *receives* functions like `add_1`, it is also a higher-order function.

Having to write simple functions like `add_1` is incredibly cumbersome. As such, languages like Python and Java make it easy to define *anonymous functions*, usually named *lambda expressions*[1]. A lambda expression in Python looks like this:

```
>>> list(map(lambda x: x + 1, [1, 2, 3]))
[2, 3, 4]
```

The idea is simple: the variable names to the left of `:` are the function's parameters, and the expression to the right of `:` is its return value. Obviously, this makes `lambda` expressions more restrictive since we cannot express multi-statement functions, but that is not the point. It provides a convenient syntax for defining short functions, which comes in handy very frequently.

Nested Functions and Closures

You have likely been introduced to the idea of a *nested function*, i.e. it is a function that is defined locally within another function. And example is as follows:

```
def f(x):
    def g(y):
        return x + y
    return g
```

`f` defines a nested local function `g`. In a sense, a nested function is just a function defined within a function. However, recall that local variables and definitions are typically erased from the *call stack* once the function has returned. Therefore, when an expression like `f(2)` is evaluated, the Python runtime should allocate a stack frame for `f`, which, internally defines `g` and has the local binding for `x = 2`. The function returns the reference stored in `g`. As the function is returned, all the local variables should have been torn down, such as the local variable `g` (however, the heap reference stored in `g` (which points to the local

function definition) is returned to the caller of `f`, so it remains in memory and is accessible). However, `x`, containing the reference to the value `2` should also be cleaned up since `x` is a local variable! In this case, how does `f(2)(3)` become `5` if the local variable `x` has been cleaned and the binding has been forgotten?

It turns out that languages that have first-class functions frequently support *closures*, that is, an environment that *remembers* the bindings of local variables. Therefore, when `f(2)` is invoked, it does not return `g` as-is, with a reference to some local `x` with no binding. Instead, it returns `g` with an environment containing the binding `x = 2`. As such, when we then invoke *that* function passing in `3` (i.e. `f(2)(3)`), it returns `x + y` where `y` is obviously `3`, but is also able to look up the environment `x = 2`, thereby evaluating to `5`.

Currying

Nested functions and closures thereby support the phenomenon known as *currying*, which is to have a multi-parameter function being converted to successive single-parameter functions. Without loss of generality, suppose we have a function `f(x, y, z)`. *Currying* this function gives us a function `f(x)`, which returns a function `g`, defined as `g(y)`, that function returns another function `h` defined as `h(z)`, and `h` does whatever computation `f(x, y, z)` does. We offer the following simple example:

```
def add(x, y, z):
    return x + y + z

# Curried
def add_(x):
    def g(y):
        def h(z):
            return x + y + z
        return h
    return g

# Simpler definition with lambda expressions
def add__(x):
    return lambda y: lambda z: x + y + z
    # the scope of lambda expressions extend as far to the right
    # as possible, and therefore should be read as
    # lambda y: (lambda z: (x + y + z))
```

Currying supports *partial function application*, which supports code re-use. You will see *many* instances of currying used throughout these notes, and hopefully this will become second-nature to you.

Parameterizing Behaviour

Consider the following functions:

```
def sum_naturals(n):
    return sum(i for i in range(1, n + 1))
def sum_cubes(n):
    return sum(i ** 3 for i in range(1, n + 1))
```

Clearly, the only difference between these two functions are the terms to sum. However, the difference in `i` and `i ** 3` cannot be abstracted into a single term. Instead, what we have to do is to abstract these as a function `f` on `i`! As such, what we want is to have a function that *parameterizes behaviour*, instead of just parameterizing values.

Since Python supports first-class functions, doing so is straightforward.

```
def sum_terms(n, f):
    return sum(f(i) for i in range(1, n + 1))
```

Then, we can use our newly defined `sum_terms` function to re-define `sum_naturals` and `sum_cubes` easily:

```
sum_naturals = lambda n: sum_terms(n, lambda i: i)
sum_cubes = lambda n: sum_terms(n, lambda i: i ** 3)
```

The process of abstracting over behaviour is no different when defining functions to abstract over data/values. Just *retain similarities and parameterize differences*! As another example, suppose we have two functions:

```
def scale(n, seq):
    return (i * n for i in seq)
def square(seq):
    return (i ** 2 for i in seq)
```

Again, we can retain the similarities (most of the code is similar), and parameterize the behaviour of either scaling each `i` or squaring each `i`. This can be written as a function `transform`, which we can use to re-define `scale` and `square`:

```
# If you notice carefully, this is more-or-less the implementation of map
def transform(f, seq):
    return (f(i) for i in seq)
scale = lambda n, s: transform(lambda i: i * n, s)
square = lambda s: transform(lambda i: i ** 2, s)
```

In fact, we can use the `transform` function to transform any iterable in whatever way we want!

Manipulating Functions

On top of partial function application and parameterizing behaviour, we can use functions to manipulate/transform functions! Doing so typically requires us to define functions that *receive* and *return* functions. For example, if we want to create a function that receives a function `f` and returns a new function that applies `f` twice, we can write:

```
def twice(f):
    return lambda x: f(f(x))

mult_four = twice(lambda x: x * 2)

print(mult_four(3)) # 12
```

As you can see, `twice` receives a function and returns a new function that applies the input function twice. In fact, we can take this further by generalizing `twice`, i.e. defining a function `compose` that performs function composition:

$$(g \circ f)(x) = g(f(x))$$

```
def compose(g, f):
    return lambda x: g(f(x))

mult_four = compose(lambda x: x * 2, lambda x: x * 2)
plus_three_mult_two = compose(lambda x: x * 2, lambda x: x + 3)

print(mult_four(3)) # 12
print(plus_three_mult_two(5)) # 16
```

This is a really powerful idea and you will see this phenomenon frequently in this course.

Specific to Python, we can use single-parameter function-manipulating functions like `twice` as decorators:

```
@twice
def mult_four(x):
    return x * 2

print(mult_four(3)) # 12
```

Although the definition of `mult_four` actually only multiplies the argument by `2`, the `twice` decorator transforms it to be applied twice, therefore multiplying the argument by `4`! While decorators are useful, Haskell does not have decorators similar to this, although, frankly, this is not a required feature in Haskell since it has features much more ergonomic than this.

Map, Filter, Reduce and FlatMap

There are several higher-order functions that are frequently used in programming. One of these functions is `map`, and is more-or-less defined as such:

```
def map(f, ls):
    return (f(i) for i in ls)
```

This is exactly what you've seen earlier in `transform`! The idea is that `map` receives a function that maps each element of the iterable `ls`, and produces an iterable containing those transformed elements. Using it is incredibly straightforward:

```
>>> list(map(lambda i: i + 1, [1, 2, 3]))
[2, 3, 4]
>>> list(map(lambda i: i * 2, [1, 2, 3]))
[2, 4, 6]
```

As you can see, `map` allows us to transform every element of an input iterable using a function. Another function, `filter`, filters out elements that do not meet a *predicate*:

```
def filter(f, ls):
    return (i for i in ls if f(i))
```

```
>>> list(filter(lambda x: x >= 0, [-2, -1, 0, 1]))
[0, 1]
```

`map` and `filter` are powerful tools for transforming an iterable/sequence. However, what about aggregations? For this, we have the `reduce` function:

```
def reduce(f, it, init):
    for e in it:
        init = f(init, e)
    return init
```

As you can see, `reduce` receives three arguments: (1) a binary operation `f` that combines two elements (the left element is initially the `init` term, and also holds every successive application of `f`, i.e. it is the *accumulator*), (2) the iterable `it`, and (3) the initial value `init`. It essentially abstracts over the *accumulator* pattern that you have frequently seen, such as a function that sums over numbers or reverses a list:

```
def sum(ls):
    acc = 0
    for i in ls:
        acc = acc + i
    return acc

def reverse(ls):
    acc = []
    for i in ls:
        acc = [i] + acc
    return acc
```

In summary, `0` in `sum` and `[]` in `reverse` acts as `init` in `reduce`; `ls` in both functions act as `it` in `reduce`; `lambda acc, i: acc + i` and `lambda acc, i: [i] + acc` acts as `f` in `reduce`. We can therefore rewrite both of these functions using `reduce` as such:

```
>>> sum = lambda ls: reduce(lambda x, y: x + y, ls, 0)
>>> reverse = lambda ls: reduce(lambda x, y: [y] + x, ls, [])
>>> sum([1, 2, 3, 4])
10
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]
```

Another way to view `reduce` is as a *left-associative fold*. To give you an example, suppose we are calling `reduce` with arguments `f`, `[1, 2, 3, 4]` and `i` as the initial value. Then, `reduce(f, [1, 2, 3, 4], i)` would be equivalent to:

```
reduce(f, [1, 2, 3, 4], i) ==> f(f(f(f(i, 1), 2), 3), 4)
```

One last function that should be unfamiliar to Python developers is a `flatMap` function, which performs `map`, but also does a one-layer flattening of the result. This function is available in other languages like Java, JavaScript and many other languages due to its connection to *monads*, but we shall give a quick view of what it might look like in Python:

```
def flat_map(f, it):
    for i in it:
        for j in f(i):
            yield j
```

The idea is that `f` receives an element of `it` and returns an *iterable*, and we loop through the elements of that iterable and yield them individually. Take for example a function that turns integers into lists of their digits:

```
>>> to_digits = lambda n: list(map(int, str(n)))
>>> to_digits(1, 2, 3, 4)
[1, 2, 3, 4]
```

If we had used `map` over a list of integers, we get a two-dimensional list of integers, where each component list is the list of digits of the corresponding integer:

```
>>> list(map(to_digits, [11, 22, 33]))
[[1, 1], [2, 2], [3, 3]]
```

If we had used `flat_map` instead, we would get the same mapping of integers into lists of digits; however, the list is flattened into a list of digits of all the integers:

```
>>> list(flat_map(to_digits, [11, 22, 33]))
[1, 1, 2, 2, 3, 3]
```

---

[1] The term *lambda expression* is inspired from the λ-calculus.

LAST UPDATED **26 OCT 2024**

Lambda Calculus

The λ calculus, invented by Alonzo Church, is, essentially, one of the simplest formal "programming" languages. It has a simple *syntax* and *semantics* for how programs are evaluated.

Syntax

Let us first consider the *untyped* λ *calculus* containing variables, atoms[1], abstractions and applications. The syntax of λ terms `e` in the the untyped λ calculus is shown here:

```
e ::= v      > variables like x, y and z
    | a      > atoms like 1, 2, True, +, *
    | λv.e   > function abstraction, such as def f(v): return e
    | e e'   > function call a.k.a function application such as e(e')
```

Part of the motivation for this new language is for expressing higher-order functions. For example, if we wanted to define a function like:

```python
def add(x):
    def g(y):
        return x + y
    return g
```

Doing so mathematically might be a little clumsy. Instead, with the λ calculus, we can write it like so:

$$\text{add} = \lambda x.\, \lambda y.\, x + y$$

Just like in Python, the scope of a λ abstraction extends as far to the right as possible, so the function above should be read as:

$$\text{add} = \lambda x.\, (\lambda y.\, (x + y))$$

We show the correspondence between terms in the λ calculus with lambda expressions in Python:

λ term	Python Expression
$\lambda x.\, x + 1$	`lambda x: x + 1`
$\lambda x.\, \lambda y.\, x\ y$	`lambda x: lambda y: x(y)`

λ term	Python Expression
$(\lambda x.\ 2 \times x)\ y$	`(lambda x: 2 * x)(y)`

Function applications are left-associative, therefore $e1\ e2\ e3$ should be read as $(e1\ e2)\ e3$, and in Python, `e1(e2)(e3)` should be read as `(e1(e2))(e3)`.

Semantics

To begin describing how λ calculus executes a program (which is really just a λ term), we first distinguish between *free* and *bound* variables in a λ term.

Definition 1 (Free Variables). A variable x in a λ term e is

- *bound* if it is in the scope of a λx in e
- *free* otherwise

Then, the functions BV and FV produce the bound and free variables of a λ term respectively. For example, $\mathrm{FV}(\lambda x.\ \lambda y.\ x\ y\ z) = \{z\}$.

Now we want to be able to perform *substitutions* of variables with terms. For example, when we have an application of the form $(\lambda x.\ e_1)\ e_2$, what we want is for e_2 to be substituted with x in e_1, just like the following function in Python:

```
def f(x): return x + 1
f(2) # becomes 2 + 1 which is 3, because we substituted x with 2
```

However, this is not straightforward because we may introduce name clashes. For example, if we had $(\lambda x.\ \lambda y.\ x\ y)y$, performing a function call with naive substitution gives us $\lambda y.\ y\ y$ which is wrong, because now the free variable x is substituted with the *bound* variable y, so the meaning is not preserved. As such, we define *substitutions* on λ terms keeping this in mind.

Definition 2 (Substitution). $e_1[x := e_2]$ is the substitution of all *free* occurrences of x in e_1 with e_2, changing the names of bound variables to avoid name clashes. Substitution is defined by the following rules:

1. $x[x := e] \equiv e$
2. $a[x := e] \equiv a$ where a is an atom
3. $(e_1\ e_2)[x := e_3] \equiv (e_1[x := e_3])(e_2[x := e_3])$
4. $(\lambda x.\ e_1)[x := e_2] \equiv \lambda x.\ e_1$ since x is not free
5. $(\lambda y.\ e_1)[x := e_2] \equiv \lambda y.\ e_1$ if $x \notin \mathrm{FV}(e_1)$
6. $(\lambda y.\ e_1)[x := e_2] \equiv \lambda y.\ (e_1[x := e_2])$ if $x \in \mathrm{FV}(e_1)$ and $y \notin \mathrm{FV}(e_2)$

$$7.\ (\lambda y.\, e_1)[x := e_2] \equiv \lambda z.\, (e_1[y := z][x := e_2]) \text{ if } x \in \mathrm{FV}(e_1) \text{ and } y \in \mathrm{FV}(e_2)$$

We give some example applications of each rule:

1. $x[x := \lambda x.\, x] \equiv \lambda x.\, x$
2. $1[x := \lambda x.\, x] \equiv 1$
3. $(x\ y)[x := z] \equiv z\ y$
4. $(\lambda x.\, x + 1)[x := y] \equiv \lambda x.\, x + 1$
5. $(\lambda y.\, \lambda x.\, x + y)[x := z] \equiv \lambda y.\, \lambda x.\, x + y$
6. $(\lambda y.\, x + y)[x := z] \equiv \lambda y.\, z + y$
7. $(\lambda y.\, x + y)[x := y] \equiv \lambda z.\, y + z$ (rename y to z before performing substitution)

The last rule where variables are renamed to avoid name clashes introduces a form of equivalence known as α congruence. It captures the idea that renaming parameters in functions does not change its meaning. For example, the two functions below are, in operation, identical:

```
def f(x):
    return x + 1
def f(y):
    return y + 1
```

In other words, if two terms differ only in the name of the bound variables, they are said to be α congruent.

Finally, we get to the actual semantics of λ calculus, which is described by β reduction. Essentially it is as we have briefly described earlier—a function application `(lambda x: e1)(e2)`, evaluates to `e1` where `x` is substituted with `e2`:

$$(\lambda x.\, e)\ y \triangleright_\beta e[x := y]$$

For example:

$$\begin{aligned}(\lambda x.\, \lambda y.\, x\ y)(\lambda x.\, x + 1)(2) &\triangleright_\beta (\lambda y.\, x\ y)[x := \lambda x.\, x + 1](2) \\ &\equiv (\lambda y.\, (\lambda x.\, x + 1)\ y)(2) \\ &\triangleright_\beta ((\lambda x.\, x + 1)\ y)[y := 2] \\ &\equiv (\lambda x.\, x + 1)(2) \\ &\triangleright_\beta (x + 1)[x := 2] \\ &\equiv 2 + 1 \\ &\equiv 3\end{aligned}$$

This is more-or-less how Python evaluates function calls:

```
>>> (lambda x: lambda y: x(y))(lambda x: x + 1)(2)
3
```

Typed Variants

Python has types, which describes the class from which an object was instantiated:

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
```

We will describe more on types in Chapter 2 (Types). But for now, know that we can also assign types to terms in the λ calculus, giving us new forms of λ calculi. The simplest type system we can add to the λ calculus is, well, simple types, forming the *simply-typed λ calculus*.

For now, we shall restrict types to be the base types to only include `int`, giving us a new language for the calculus:

```
Terms
e ::= v          > variables like x, y and z
   |  a          > atoms like 1, 2, +, *
   |  λv: t.e    > function abstraction, such as def f(v: t): return e
   |  e e'       > function call a.k.a function application such as e(e')

Types
t ::= int          > base type constants, only including integers
   |  t -> t'      > type of functions; -> is right-associative
```

The introduction of types to the calculus now adds the notion of *well-typedness* to the language. Specifically, not all terms in the untyped λ calculus are well-typed in the simply typed λ calculus. To formalize this notion of well-typedness, we define *typing rules* that dictate when a term is well-typed, and what type a term has.

First we have *typing environments* $\Gamma, \Delta, \ldots$, which are sets (or sometimes lists) of *typing assumptions* of the form $x : \tau$, stating that we are assuming that x has type τ. Then, the *typing relation* $\Gamma \vdash e : \tau$ states that in the context Γ, the term e has type τ. The reason we need typing environments is so that the types of in-scope bound variables in λ terms are captured and can be used in the derivation of the types of terms. Instances of typing relations are known as *typing judgements*.

The validity of a typing judgement is shown by providing a *typing derivation* that is constructed using *typing rules*, which are inference rules:

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B}$$

Which basically states that if all the statements A_i are valid, then the statement B is also valid.

Then, the simply-typed λ calculus uses the following rules.

1. If a variable x has type τ in Γ then in the context Γ, x has type τ

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

2. If an atom a has type τ then we can also judge the type of a accordingly

$$\frac{a \text{ is an atom of type } \tau}{\Gamma \vdash a : \tau}$$

3. Abstraction: If in a certain context we can assume that x has type τ_1 to conclude e has type τ_2, then the same context without this assumption shows that $\lambda x : \tau_1 . e$ has type $\tau_1 \to \tau_2$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2}$$

4. Application: If in a certain context e_1 has type $\tau_1 \to \tau_2$ and e_2 has type τ_1, then $e_1\ e_2$ has type τ_2

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

These rules can be used to perform *type checking* (the procedure of checking the well-typedness of a term) or *type reconstruction* (the procedure of finding the types of terms where their typing information is not present, as is the case in the untyped λ calculus).

For example, in our calculus we can show that $\lambda x : \texttt{int} \to \texttt{int} . \lambda y : \texttt{int} . x\ y$ has type $(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$ and is therefore a well-typed term:

$$\frac{x : \texttt{int} \to \texttt{int} \in \Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int}}{\Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash x : \texttt{int} \to \texttt{int}}$$

$$\frac{y : \texttt{int} \in \Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int}}{\Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash y : \texttt{int}}$$

$$\frac{\Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash x : \texttt{int} \to \texttt{int} \qquad \Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash y : \texttt{in}}{\Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash (x\ y) : \texttt{int}}$$

$$\frac{\Gamma, x : \texttt{int} \to \texttt{int}, y : \texttt{int} \vdash (x\ y) : \texttt{int}}{\Gamma, x : \texttt{int} \to \texttt{int} \vdash (\lambda y : \texttt{int} . x\ y) : \texttt{int} \to \texttt{int}}$$

$$\frac{\Gamma, x : \texttt{int} \to \texttt{int} \vdash (\lambda y : \texttt{int} . x\ y) : \texttt{int} \to \texttt{int}}{\Gamma \vdash (\lambda x : \texttt{int} \to \texttt{int} . \lambda y : \texttt{int} . x\ y) : (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}}$$

That means the following lambda expression in Python (assuming only `int` exists as a base type) will have the same type:

```
>>> f = lambda x: lambda y: x(y) # (int -> int) -> int -> int
>>> my_fn = lambda x: x + 1 # int -> int
# f(my_fn): int -> int
# f(my_fn)(3): int
>>> f(my_fn)(3) # int
4
>>> type(f(my_fn)(3))
class <'int'>
```

---

[1] The actual untyped λ calculus does not have atoms like numbers, booleans etc. However, for simplicity's sake we shall include them in the language. The version we present is frequently termed the *applied* λ *calculus*, in contrast with usual presentations known as the *pure* λ *calculus* which omits atoms.