Haskell Tutorial

A guide to Haskell basics

Ruben Saunders

Contents

1 Introduction

Haskell is a purely functional programming language. The following sections will give a brief overview of Haskell, and how to install it.

1.1 Language Overview

In Haskell, everything is a *pure* function - that is, they abide by the Mathematical definition of a function; they map inputs to a unique output.
Data is immutable, meaning that our data types cannot be changed in-place. Combined, this means that there a re few or no side-effects from functions, which make programming more simple.
Haskell is declarative, meaning that the program defines what the issue is, rather than simply giving an algorithm to solve a problem.
Functional programs are easier to verify as we can use maths to verify an algorithm.

1.2 Installation

Link: `https://www.haskell.org/ghcup/`
I used GHCup to install several components of the Haskell toolchain.

1.2.1 The Haskell Toolchain

The Haskell Toolchain consists of several useful tools for Haskell compilatio and development:

- **GHC** - the Glasgow Haskell Compiler;
- **cabal-install** - Cabal installation tool for managing Haskell software;
- **Stack** - a cross-platform proram for developing Haskell projects;
  - **Msys2** - provides a UNIX shell and environment which is necessary for executing configuration scripts.
- **haskell-language-server** - a language server which may be integrated into an IDE;

1.2.2 Install Command

The command to use on Windows (in a normal Powershell instance) is

```
Set-ExecutionPolicy Bypass -Scope Process -Force;[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; try { Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Invoke-WebRequest https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 -UseBasicParsing))) -ArgumentList $true } catch { Write-Error $_ }
```

2 Evaluation

This section will explain how Haskell evaluates its expressions and is fundamental to understanding how the language operates.
In summary, Haskell employs *lazy evaluation* which uses *outermost reduction* and *sharing*.

2.1 Lazy Evaluation

Haskell is lazy. This means that it will only evaluate an expression when that expression is required; otherwise, the expression is stored in an un-evaluated form as a *thunk*.
An un-evaluated expression may be viewed in GHCi using `:sprint <expr>`. Underscores represent thunks which have not been evaluated.
To evaluate an expression, Haskell recursively *reduces* terms and applies atomic calculations where necessary (operations such as + and ⋆ on numeric types).

2.2 Reductions

This is the technique Haskell uses to evaluate expressions. The idea is that anything appearing on the left-hand side of = may be replaced by its right-hand side (and vice versa). Pattern matching is used in more complex replacements.
Example:

```
square :: Int -> Int
square x = x * x

f :: Int -> int -> Int
f x y = y + square x

155 + f 94 10            -- Apply definition of 'f'
= 155 + 10 + square 94   -- Apply definition of 'square'
= 155 + 10 + 94 * 94     -- Atomic operation '(*)'
= 155 + 10 + 8836        -- Atomic operation '(+)'
= 155 + 8846             -- Atomic operation '(+)'
= 9001                   -- Fully evaluated
```

As Haskell functions are mathematically pure, the order of reduction does not matter- the end result is the same. We could reduce `155 + 10` before applying `square` and the result would still be the same. This is known as the *Church-Rosser* theorem and applies to some variants of lambda-calculus.
There are two reduction variants.

2.2.1 Innermost Reduction

Fully evaluate the arguments of a function *before* the function definition is applied.

```
fst (square 3, square 4)
= fst (3*3, square 4)    -- Apply 'square'
= fst (9, square 4)      -- Atomic operation '(*)'
= fst (9, 4*4)           -- Apply 'square'
= fst (9, 16)            -- Atomic operation '(*)
= 9                      -- Apply 'fst'
```

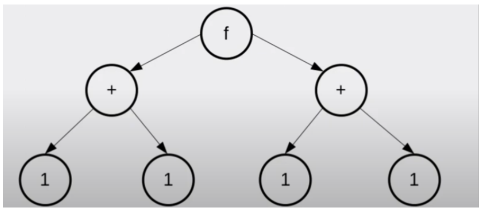
2.2.2 Outermost Reduction

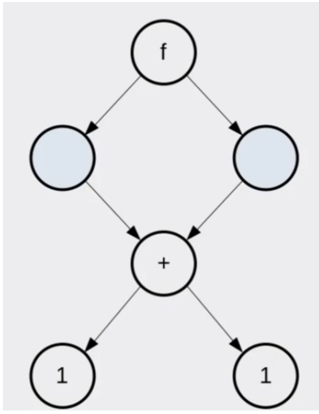
Fully apply function definitions first.

```
fst (square 3, square 4)
= square 3                -- Apply 'fst'
= 3*3                     -- Apply 'square'
= 9                       -- Atomic operation '(*)'
```

2.2.3 Graph Reduction

This is all good in theory, but how does Haskell *actually* evaluate a program? Under the hood, haskell uses graph reduction. Essentially, every expression and equation in Haskell is part of a large graph structure. Each vertex contains either a function which can be reduced, or an atomic value. To deal with sharing, each non-value vertex points to a placeholder vertex where the result will be stored, which i turn points to the expression. Therefore, *sharing* is where two or more placeholder vertices point to the same expression.





2.3 Sharing

Possible due to lack of side effects, sharing results in any equivalent thunks only being evaluated once. This is done by *memoising* evaluated thunks to avoid re-evaluation of equivalent thunks in the future.

```
(1 + 1) + 2 + (1 + 1)
= (2) + 2 + (2)
= (2) + 4
= 6
```

2.4 Normal Form

An expression is in a normal form iff it is fully evaluated (i.e. no further reductions can be applied). Examples:

```
1 + 1 => 2
(1, 2+2) => (1, 4)
[True && False] => [False]
(\x -> x * 10) -- Already in normal form!
```

Note that the following are not in normal form:

- *Function definitions.* Expand the definition into a value.

```
f x = ...
-- Becomes
f = \x -> ...
```

- *Top-level pattern matching.* Expand into a `case .. of ..` expression.

```
f <p1> = <e1>
f <p2> = <e1>
-- Becomes
f = \x -> case x of
    p1 -> e1
    p2 -> e2
```

2.4.1 Weak Head Normal Form

An expression which is fully evaluated *up to* ***at least*** the first data constructor. Values/equations in WHNF are:-

1. A data constructor
2. A built-in function applied to too few arguments e.g. `(+ 2)`
3. A lambda expression

```
1 + 1 => 2
(1, 2+2) -- In WHNF as values are inside a data constructor
[True && False] => _ : _ -- The values do not matter. Only the constructor '(:)'
    matters in WHNF.
```

The `seq` function is *strict* in its first argument - it enforces its first argument to be evaluated to WHNF. This function is defined this way explicitly in GHC and is one of the only functions to do this.

3 Variables

Variables are name-bounded values. Variables in Haskell are immutable - they cannot be changed.

```
name = <value>
```

Variable types are *inferred*. To explicitly assign variables a type,

```
name :: Type
```

In GHCI, the type of a symbol may be retriesed by `:t name`.

3.1 Local Name Binding

Two methods are provided to bind a symbol to a value in a local scope: `let` and `where`. Each follow a "main" expression.

3.1.1 let

Syntax:

```
let
  <symbol> = <expr>
  <symbol2> = <expr2>
  ...
in
<main_expr>
```

For example, re-define `in_range` as follows:

```
in_range x min max =
  let
    in_lb = min <= x
    in_ub = max > x
  in
  in_lb && in_ub
```

3.1.2 where

Syntax:

```
<main_expr>
where
  <symbol> = <expr>
  <symbol2> = <expr2>
  ...
```

For example, re-define `in_range` as follows:

```
in_range x min max = in_lb && in_ub
  where
    in_lb = min <= x
    in_ub = max > x
```

4 Functions

Functions are defined to map a value from an input set - the *domain* - into an output set - *the co-domain*. Every output of the function is contained in a subset of the co-domain, called the *range*. **Pure** mathematical functions must be able to map every value from the domain, and each input value must map to only one output value.
In Haskell, occurences of functions are expanded into their RHS.

4.1 Function Definition

Functions are defined by providing its name, a list of arguments, and setting it equal to an expression.

```
name arg1 arg2 ...argn = <expr>
```

The arguments may be set to constants, or may be given a name to accept a variable value.
The type of a function is specified by an arrow (->) seperated list of its argument types and its return type:

$$\texttt{func} :: \ \texttt{type}_1 \ \texttt{->} \ \texttt{type}_2 \ \texttt{->} \ \texttt{...->} \ \texttt{type}_n \ \texttt{->} \ \texttt{type}_{\texttt{return}}$$

For an example, take a function which returns the sum of the elements of an array:

```
sum :: [a] -> Int -- Takes an array of arbitrary type and returns an integer
sum [] = 0 -- Define the sum of an empty array to be zero
sum (h:t) = h + sum t -- Define the sum of an array to be the head plus the sum
  of the tail
```

4.1.1 Infix Functions

A good example of infix functions are operators such as +. Functions which take two arguments may be writen between the arguments instead.
For example, say we had add a b = a + b. Then add 5 7 and 5 \`add\` 7 are equivalent.
To reference the function defined by an operator i.e. +, surround it with parenthesis i.e. (+).
The associativity and precedence of infix operators may be changed via infixr/infixl <prec> <function name> where $0 < \texttt{prec} \leq 9$.

4.1.2 Pattern Matching

Parameters of functions may be matched against a pattern. Note that order matters: more specific patterns should be defined first, with general cases last.

- Accept any value by using a symbol e.g. f x = ...
- Accept a certain value e.g. f 0 = ..., g [] = ...
- List splicing using (x:xs) where x is the head, xs is the tail.
- Tuple unpacking e.g. (Int,Int) could be unpacked using (x,y).
- Accept and extract a custom types. E.g., say we had type Pos = (Int, Int), we would define getX (x, y) = x.
- Accept and extract a custom datatype. E.g., say we had data Num = Zero | Succ Num. We could then define f Zero = ... and f (Succ n) = ...

Pattern matching may also be done using case ... of ... construct.

4.2 Function Application

A function is applied (called) to some arguments as follows:

```
name arg1 arg2 ...argn
```

For example, consider the function in_range x min max = x >= min && x < max, an implementation of $x \in [min, max)$.
Then, in_range 3 0 5 would evaluate to True, but in_range 5 0 5 would evaluate to False.

4.3 Recursion

Recursion is the process of a function calling itself. Recursion requires a *base case* to stop the function recursing indefinitely.
There are many ways to implement recursion, which will be demonstrated using the *factorial*, defined as

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1 = \prod_{k=1}^{n} k$$

4.3.1 Defined Base Case

We can hard-code the case where the function is called with the base case:

```
  fac 1 = 1
  fac n = n * fib (n-1)
```

4.3.2 If-Else Expression

We can use the if-else expression:

```
if <expr> then <ifTrue> else <ifFalse>
```

For example,

```
  fac n = if n <= 1 then 1 else n * fac (n-1)
```

4.3.3 Guards

Guards are similar to piece-wise functions.

```
<main_expr>
  | <expr> = <value>
  | <expr2> = <value2>
  ...
  | otherwise = <default_value>
```

Where <expr> is a boolean expression. If <expr> is matches, then <value> will be returned. If none is matched, the otherwise is returned.
For example,

```
  fac n =
    | n <= 1     = 1
    | otherwise = n * fac (n-1)
```

4.3.4 Accumulators

In this example, we define an auxiliary function `aux` inside `fac` to calculate the the factorial

```
fac n = aux n 1
  where
    aux n acc
      | n <= 1     = acc
      | otherwise = aux (n-1) (n*acc)
```

This is called *tail recursion*. This is because the final result of `aux` is the result we want, meaning that it is much more memory efficient. A good compiler could even unwind this into a non-recursive imperative approach using a loop. (For more insight, see `https://www.youtube.com/watch?v=_JtPhF8MshA`.)
Normal recursion (using an above definition of `fac`):

```
fac 4
= 4 * fac 3
= 4 * (3 * fac 2)
= 4 * (3 * (2 * fac 1))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

Tail recursion (using the definition in this sub-section):

```
fac 4
= aux 3 4
= aux 2 12
= aux 1 24
= 24
```

4.4 Lambdas

Syntax:

```
(\<args> -> <expr>)
```

Some examples:

- `(\x -> x+1) 2` returns `3`.
- `(\x y z -> x+y+z) 1 2 3` returns `6`.

Lambdas may be bound to names.

4.5 Higher Order Functions

Higher order functions are functions that take other functions as arguments.
For example, a function which takes another function and applies it to an argument:

```
app :: (a -> b) -> a -> b
app f x = f x
```

A synonym of such a function is the dollar (`$`) operator: `($) :: (a -> b) -> a -> b`.

4.5.1 Useful Higher Order Functions

- `map :: (a -> b) -> [a] -> [b]` applies a function to every element on an array.

  $$L' = \{f(x) : x \in L\}$$

  Example: `map (\x -> x^2) [1,2,3]` returns `[1,4,9]`.

- `filter :: (a -> Bool) -> [a] -> [a]` filters the list on a predicate.

$$L' = \{x \in L : P(x)\}$$

  Example: `filter (\x -> mod x 2 == 0) [1,2,3,4,5]` returns `[2,4]`.

- `fold :: (a -> b -> b) -> b -> [a] -> b` processes a list with some function to produce a single value, starting at a given value.

  In Haskell, `fold` doesn't exist, but rather `foldr` and `foldl` wherein the position of the base value is different

```
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
-- foldr f a [x1, x2, ..., xn] = (x1 `f` (x2 `f` (... (xn `f` a))))

foldl f a = []
foldl f a (x:xs) = f (foldl f a xs) x
-- foldl f a [x1, x2, ..., xn] = (...((a `f` x1) `f` x2) `f` xn)

```

  See `code/folding.hs`

  Example: `foldr (+) 0 [1,2,3,4,5]` returns `15`.

4.6 Currying

The principle behind currying is that given

`f :: a -> b -> c -> d`

We could re-write this as

`f' :: a -> (b -> (c -> d))`

For example, one could define a function `add` in multiple ways:

```
add x y = x + y
add x = (\y -> x + y)
add = (\x -> (\y -> x + y))
```

4.6.1 Currying

```
curry :: ((a,b -> c) -> a -> b -> c)
curry f x y = f (x,y)
```

4.6.2 Uncurrying

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

4.6.3 Partial Function Application

Using the last definition of `add`, consider the result of `add 1`. This would be a new function; `add 1 :: Int -> Int`. This is known as a *section*.
A good example would be using `map`.

`doubleList = map (\x -> x * 2)`

4.7 Function Composition

Function composition is a way to combine functions. For this, we use the dot `(.)` operator.

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

Then, `(f.g)` is equivalent to `(\x -> f (g x))`.
For example, all three definitions of `descSort` are equivalent:

```
descSort = reverse . sort
descSort = (\x -> reverse (sort x))
descSort = reverse (sort x)
```

5 Types

Every expression in Haskell has a type. Types are inferred, even when given explicitly.
Types always begin with an UPPERCASE letter.
The cardinality of a type is how many *states* the data type could hold. Polymorphic types have no cardinality.

5.1 Variable Types

```
var :: type
```

5.1.1 Lists

To define a list of a `type`, one would write `[type]`. This may be nested.

5.2 Function Types

```
func :: type1 -> ... -> typeN -> ret_type
```

Where the function `func` takes n arguments of types `type1, ..., typeN` and returns `ret_type`.

5.2.1 Type Variables

Type variables may be used where any type would be permissable and must be lowercase. For example,

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

This is called "parametric polymorphism".

5.3 Type Aliasing

This doesn't define a new datatype, but rather an alias for another type.

```
type Pos = type
```

A common example is using a tuple e.g. `type Pos = (Int, Int)`. The type name need not be stated in pattern matching.

5.4 Type Classes

Type classes may be used to restrict the types a polymorphic function may take. This is useful if we would like to use features in a polymorphic function that may only be available to certain types. For a type to be a member of a type class, it must implement all of the required methods.
To impose a constraint on variable `a` in a function `f`: `f :: (<TypeClass> a, ...) => ...`.
This is called "ad-hoc polymorphism".

5.4.1 Definition

A type class definition begins with
`class <Name> <var> where`
Below is a list of function declarations. For a type to be a member of `<Name>`, it must implement all of these functions.

5.4.2 Implementation

To define a new type which belongs to a type class: `instance <TypeClass> <TypeName> where`
Where below this is a list of function definitions.

5.4.3 Common Type Classes

Common type classes include:

- `Eq` – types which may be compared i.e. `(==)` is defined;
- `Num` – numeric types, gives us access to standard mathematical operations i.e. `(+), (-), (*), abs, ...` are defined;
- `Ord` – types which may be ordered, imposes a total ordering i.e. `(<), (>), (<=)` are defined;
- `Read` – types which may be converted from a string i.e. `read` is defined;
- `Show` – types which may be converted to a string i.e. `show` is defined;
- `Integral` – types which are integer-like i.e. `div, ...` is defined;
- `Floating` – types which are float-like i.e. `(/), ...` is defined;
- `Enum` – types which may be enumerated i.e. `succ, pred, ...` are defined;

N.B. for information on complex type classes, see Type Class section

5.4.4 Instances

Intances allow us to write functions which make use of type classes. Syntax:
`instance (<constraints>) => <typeClass> <value> where` followed by a list of function definitions.

5.4.5 Derivation

The `deriving` keyword can be used to automatically generate implementations for the given type class(es).
Syntax: `data <Name> = ...deriving (<Class1>, ...)`
Example: `data Shape = Circle Int | Rect Int Int deriving Show`. Then, `print (Circle 5)` → `Circle 5`.

5.5 Defining Datatypes

The data keyword is used to define a new datatype; unlike the above, these are entirely custom.
This may be done using the `data` keyword:

```
data Name = Constructor1 [<args>] | ...
```

where `<args>` are the *types* of each argument, not literals.
Constructors are either plain values, or functions which take `args` and return the datatype.
Data consturctors can include polymorphism by including type variables after `<Name>` e.g. `data Maybe a = Nothing | Just a`.

5.5.1 Examples

- `rock-paper-scissors.hs` – A basic example revolving around Rock-Paper-Scissors;
- `expr.hs` – A program to build and evaluate expressions;
- `tree.hs` – A representation of a tree structure;
- `nat-num.hs` – A definition of natural numbers using the successor function;

5.6 Records

Records allow data to be stored with an associated name.
Syntax:

```
data <Name> = <Name> { <field> :: <type>, ...}
```

This will automatically generate functions <field> :: <Name> -> <type> to extract said properties. This has the side-effect that field names must be globally unique.

5.6.1 Multiple Constructors

Note that records may also have mutliple constructors,

```
data Point = D2 { x :: Int, y :: Int } | D3 { x :: Int, y :: Int, z :: Int }
```

Duplicate field names in this context is OK.
This will generate functions `x`, `y` and `z` all with the signature `x/y/z :: Point -> Int`. Both `x` and `y` will work on either `D2` or `D3`, but applying `z` to `D2` will throw an exception.
For an example, see `code/vector.hs`.

6 Useful Types

This section will list some common, useful types which should be known.

6.1 Maybe

The `Maybe` type is incredibly useful, as it can be used to represent the *absence* of a value. This is useful when our function is passed invalid data, for example.
If it defined as: `data Maybe a = Nothing | Just a`

Functions Found inside `Data.Maybe`.

- `isJust :: Maybe a -> Bool` – returns if the passed `Maybe` is a `Just` value;
- `fromMaybe :: a -> Maybe a -> a` – returns the `Just` value if a value is present, else returns the default value;
- `fromJust :: Maybe a -> a` – returns the `Just` value, or throws an exception if recieved `Nothing`;
- `catMaybes :: [Maybe a] -> [a]` – returns a list containing all the `Just` values.
- `mapMaybe :: (a -> Maybe b) -> [a] -> [b]` – maps a function over a list of `Maybe`s, discarding any `Nothing`s;
- `maybe :: b -> (a -> b) -> Maybe a -> b` – takes a maybe. If `Just`, applies a function and returns. Else, returns a default;

6.2 Either

The `Either` type can be used to represent a union of types – a value which is either one type, or another.
It is defined as `data Either a b = Left a | Right b`

Functions Found inside `Data.Either`.

- `lefts :: [Either a b] -> [a]` – returns an array of left-hand side values;
- `rights :: [Either a b] -> [b]` – returns an array of right-hand side values;
- `isLeft :: Either a b -> Bool` – returns whether the provided value is a left-hand side value;
- `isRight :: Either a b -> Bool` – returns whether the provided value is a right-hand side value;
- `fromLeft :: a -> Either a b -> a` – returns the left-hand side value if a `Left` is provided, else returns a default;
- `fromRight :: n -> Either a b -> n` – returns the right-hand side value if a `Right` is provided, else returns a default;
- `either :: (a -> c) -> (b -> c) -> Either a b -> c` – processes the left- or right-hand value in the `Either` as per the given functions;
- `partitionEithers :: [Either a b] -> ([a],[b])` – traverses the list, placing and `Left` values in one list and any `Right` values in another;

Use – Error Handling `Either` can be used in place of `Maybe` for error handling.
`Left` can represent a correct input/output with `Right` holding the incorrect value.

7 Collections

Haskell has two collections: lists, and tuples.

7.1 Lists

A *mutable* collection of elements of the *same type*. Every elements has an ordinal.
A list of type `type` has the given type signature

`name :: [type]`

7.1.1 Construction

A list may be greated by the following constructor:

- Using square brackets: `[`x_1`, `x_2`, ..., `x_n`]`
- Using the prepend opeator: $x_1 : x_2 : \ldots : x_n$ `:[]`. With the syntax of `x:list`, it prepends `x` to the list `list`.

7.1.2 Pre-defined functions

Many pre-defined functions for lists are defined in the `Data.List` module.

General Functions These functions work on a list of any type, namely `[a]`.

- `head <list>`. This function returns the head (x_1) of the list. Example: `head [1,2,3]` returns `1`.
- `tail <list>`. This function returns the tail of the list. Example: `tail [1,2,3]` returns `[2,3]`.
- `length <list>`. This function returns the length of the list. Example: `length [1,2,3]` returns `3`.
- `init <list>`. This function returns the list without the last element. Example: `init [1,2,3]` returns `[1,2]`.
- `null <list>`. This function returns whether the list is empty. Example: `null [1,2,3]` returns `False`.
- `take <n> <list>`. This function returns a list of the first n elements of the list. Example: `take 2 [1,2,3,4,5]` returns `[1,2]`.
- `drop <n> <list>`. This function returns a list excluding the first n elements of the list. Example: `drop 2 [1,2,3,4,5]` returns `[3,4,5]`.
- `<list1> ++ <list2>`. This function "append" returns a concatenation of both lists. Example: `[1,2] ++ [3,4]` returns `[1,2,3,4]`.

Boolean Functions These functions are of the type `fn :: [Bool] -> Bool`

- `and <list>`. This functions returns `True` if every elements in `<list>` is `True`.
- `or <list>`. This functions returns `True` if at least one element in `<list>` is `True`.

7.1.3 List Comprehension

List comprehension can be used to transform one or more lists according to a predicate. Syntax:

```
[ <gen> | <elem> <- <list>, ..., <guard>, ...]
```

Where

- <elem> <- <list> is called a *generator* – it binds each value from <list> to <elem> in turn so that they may be used. There may be multiple generators, in which case they will be worked through left-to-right.
- <guard> is a statement which returns a Bool. If false, the current bound value(s) will not be output.

Examples:

- [2*x | x <- [1,2,3]] generates [2,4,6]
- [x^2 | x <- [1,2,3], x > 1] generates [4,9]
- [(x,y) | x <- [1,2,3], y <- ['a','b']] generates [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]

7.1.4 Ranges

Generate ranges (arithmetic sequences) in Haskell using the ellipse ..:

```
[<start>, [<step>] .. [<end>]]
```

Where

- step is optional, and default to 1 e.g. [1..5] = [1,2,3,4,5], [1,3..5] = [1,3,5]
- <end> may be omitted to generate an infinite list e.g. [1 ..] = [1,2,3,...].

7.2 Tuples

An *immutable* collection of elements of *different types*.
A tuple has the signature

$$\texttt{(x,y,...)} \;::\; \texttt{(type}_x\texttt{, type}_y\texttt{, ...)}$$

8 Kinds

The types of *types* and *type constructors* are called *kinds*.
A kind signature is denoted by `::`. The number of *kinds* correspond to the number of type parameters a type constructor takes.

- Types which take no type parameters are of kind $\star$ ("*type*").
  - `Bool :: *`
  - `String :: *`
  - `Maybe :: * -> *`
- When we apply a type parameter, one of the parameters is consumed in the kind
  - `Maybe Int :: *`

9 Modules

In Haskell, each file is a module. Hence, each file (other than the entry file) must begin with a module declaration:

```
module filename [(n1, n2, ...)] where
```

By default, every symbol is exported. If `(n1, n2, ...)` is included, only symbols `n1`, `n2` etc. are exported.

9.1 Importing

To import a module, use an import statement:

```
import module [(n1, n2, ...)]
```

By default, every symbol exported by `module` is imported. If `(n1, n2, ...)` is included, only symbols `n1`, `n2` etc. are imported.
To import `Animals.hs` one would write `import Animals`. To import `Farm/Tractor.hs` one would write `import Farm.Tractor`.
Once imported, symbols may be used freely. For example, if the function `double` is imported, to reference it we would write `double`. However, if two different definitions for `double` are imported, we must use its full name e.g. `Module.double`.

9.1.1 Qualified Imports

Syntax:

```
import qualified module [(...)]
```

This forces the module name to precede any symbols imported. In the example above, `Module.double` **must** be used.

9.1.2 Aliased Imports

Syntax:

```
import module [(...)] as alias
```

Using the above example, now, instead of writing `Module.double` one would now write `Alias.double`.

9.1.3 Hiding Imports

The `hiding` keyword can be used to omit imports. E.g. `import Prelude hiding (map)` would import every function from `Prelude` but omit `map`.

9.2 Importing Datatypes

Let's say `data DataType = A | B | C` is defined inside module `Module`.

- To import every constructor, we'd write `import Module (DataType(..))`
- To import only constructor `A`, we'd write `import Module (DataType(A))`
- To import constructors `A` and `B`, we'd write `import Module (DataType(A,B))`

10 Type Classes

This section covers more complex type classes. For basics, see `Types -> Type Classes`.

10.1 Semigroups

A type is a Semigroup if there exists some function which, when two values from the group are combined, that value is also in the Semigroup.

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

infixr 6 <>
```

This diamond operator must be **associative**:

```haskell
x <> (y <> z) == (x <> y) <> z
```

10.2 Monoid

A Monoid is an extension of a Semigroup, adding an identity element.

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

  -- Optional functions
  mappend :: a -> a -> a
  mappend = (<>)

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

`mempty` must be an identity element for the Semigroup. Therefore, it must obey:

```haskell
x <> mempty == x
mempty <> x == x
```

10.2.1 Multiple Monoids

A type may have multiple Monoid implementations (i.e. multiple viable operators exist which satisfy Monoidal conditions).
For example, take numerical types:

```haskell
newtype Sum n = Sum { getSum :: n }
instance Num a => Semigroup (Sum a) where
  (<>) = (+)
instance Num a => Monoid (Sum a) where
  mempty = 0


newtype Product n = Product { getProduct :: n }
instance Num a => Semigroup (Product a) where
  (<>) = (*)
instance Num a => Monoid (Product a) where
  mempty = 1
```

10.2.2 Foldable

A Foldable datatype is one which may be reduced to a single value i.e. folded in on itself.

```haskell
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

10.3 Functors

A Functor is a type class which may have an operation mapped over it.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b -- Infix symbol is <$>

  -- The following functions are optional
  (<$) :: a -> f b -> f a
  (<$) = fmap . const
```

A Functor instance must obery the following laws:

- The mapping must preserver the structure of the arguments.
- **Identity**:

```
fmap id == id
```

- **Distributive over Composition**:

```
fmap (f . g) == fmap f . fmap g
```

10.3.1 Fmap

`fmap` allows a function to be mapped over a structure without the internal structure of the Functor changing.
Example:

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

instance Functor Tree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node a b c) = Node (fmap f a) (f b) (fmap f c)
```

10.4 Applicatives

Applicatives are Functors with more and better functionality, with `<*>` essentially *injecting* a value into a wrapped function, and `pure` allowing easy construction of an applicative.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

  -- Optional functions
  (*>) :: f a -> f b -> f b
  a *> b = b
  (<*) :: f a -> f b -> f a
  a <* b = a
```

An Applicative must obey the following laws:

- **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)
```

10.5 Monads

A Monad allows the transformation of a value into a Monad via a function.

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- "Bind"

  -- Optional functions
  (>>) :: m a -> m b -> m b -- "Then"
  a >> b = a >>= \x -> b

  return :: a -> m a
  return = pure
```

Any implementation must abide by these laws:

- **Left identity**:

```
return a >= h == h a
```

- **Right identity**:

```
m >= return == m
```

- **Associativity**:

```
(m >>= g) >>= h == m >>= (\x -> g x >>= h)
```

10.5.1 Bind

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

This function takes a monad and a function which takes a raw value and returns a new monad, and returns another new monad.
When implemented, then, we may vary the action taken depending on the value of the provided monad, such as returning a default value – this is what `(>>=)` does with `Maybe`, as shown below:
Example:

```
  add :: Num a => Maybe a -> Maybe a -> Maybe a
  add mx my = mx >>= (\x -> my >>= (\y -> Just (x + y)))

  -- Then addition works as expected
  add (Just 1) (Just 2) -- => Just 3
  -- And if either one of the arguments is Nothing, it returns Nothing
  add Nothing (Just ?) -- => Nothing
```

10.5.2 Then

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

This function discards the second monad given to it. `m >> n` is equivalent to `m >>= \_ -> n`.
"Then" can be though of wanting to carry out an action but not caring what the result is.

10.5.3 Return

```
return :: Monad m => a -> m a
```

Return wraps a monad around a raw value.
Using the example from the Bind section, we could substitute the explicit `Just` with the more general `return`. Now, this would theoretically work with any appropriately-defined monad.

```haskell
add mx my = mx >>= (\x -> my >>= (\y -> return (x + y)))
```

10.5.4 Fail

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

Fail is intended to be called when something goes wrong. The default implementation is to call `error` (i.e. error out of the program), but it may be implemented so that certain errors may be handled and return an appropriate monad as a response.

10.5.5 "do" Syntax

Chaining together applications of `(>>=)`, `(>>)` and lambda functions can get tedious; that's where the syntactic sugar "do" expression comes in.
The "statements" inside of `do` are executed in order, and if one "statement" fails this will be propagated through.

- **Bind**

```haskell
m >>= \x -> ...
-- Becomes
do
  x <- m
  ...

```

- **Then**

```haskell
m >> ...
-- Becomes
do
  m
  ...

```

Example:

```haskell
-- Re-writing the above definition of 'add'
add mx my = do
  x <- mx
  y <- my
  return $ x + y
```

10.5.6 Kleisli-Composition

This operator, denoted >=>, acts as a Monad composition operator. It is defined in `Control.Monad` as so

```haskell
infixr 1 >=>
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >=> g = \x -> f x >>= g
```

11 I/O

I/O produces an issue with Haskell as I/O functions aren't *pure*.

11.1 The IO Type

All I/O functions in Haskell have the following type: IO <value>.
This special type holds a given I/O action. When the IO value is used, the stored action will be carried out, and IO <value> is returned.
For example, in GHCI

```
> hi = putStrLn "Hello, World!"
> hw
Hello, World!
```

Notice how nothing was outputted until the IO value was used. Note that hw may be used mutliple times.

11.2 Input

- getLine :: IO String – retrieves a line of input from STDIN;
- readLn :: Read a => IO a – retrieves a line of input from STDIN, reading it as specified by a;

11.3 Output

- putStr :: String -> IO () – puts the given string to STDOUT;
- putStrLn :: String -> IO () – puts the given string to STDOUT on a new line;
- print :: Show a => a -> IO () – essentially the same as putStrLn . show;

11.4 Extracting <value>

IO is a *monad*, and should be extracted as such.

```
greet :: IO ()
greet = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

You can only extract values from IO inside of another IO action.
For a more complex example, see code/IO.hs.

11.5 Environment

The following functions are defined in System.Environment.

11.5.1 Command-Line Arguments

Command-line arguments are arguments passed to the executable e.g. ./prog.exe arg1 arg2 ...
These can be accessed via getArgs :: IO [String]
Note, the program name "prog.exe" is ommited; this can be access via getProgName :: IO String

11.5.2 Environment Variables

The function `getEnvironment :: IO [(String, String)]` gets a list of all environment variables in name-value pairs.
To get only one variable, the function `lookupEnv :: String -> IO (Maybe String)` returns the value of an environment variable.
The function `withArgs :: [String] -> IO a -> IO a` loads sets the environment variables inside an IO action.

11.5.3 Error Handling

Defined in `System.Exit`

- `exitWith :: ExitCode -> IO a` exits the program with the provided exit code (`ExitCode = ExitSuccess | ExitFailure Int`);
- `exitSuccess :: IO a` exits the program with exit code of success;
- `exitFailure :: IO a` exits the program with exit code of failure (1);
- `die :: String -> IO a` prints the given message, then exits the program with exit code of failure (1);

11.6 Files

The symbols `stdout :: Handle` and `stdin :: Handle` are handles to the process' input/output streams. I/O functions defined above use the following functions with these handles provided.

- `readFile :: FilePath -> IO String` – reads contents of the file
- `writeFile :: FilePath -> String -> IO ()` – writes to the file (overwrites contents if exists)
- `appendFile :: FilePath -> String -> IO ()` – appends to the file
- `renameFile :: FilePath -> FilePath -> IO ()` – renamed the given file to the second argument
- `deleteFile :: String -> IO ()` – deletes the given file

Alternatively, you can use file handles.

- `openFile :: FilePath -> IOMode -> IO Handle` – Open a file in the given mode.
  ```
  data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
  ```
- `hGetContents :: Handle -> IO String` – Get contents of the file
- `hPutStr :: Handle -> String -> IO ()` – writes the given string to the file
- `hPrint :: Show a => Handle -> a -> IO ()` – converts a to a string and write to the file
- `hClose :: Handle -> IO ()` – Close the given handle
- `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a` – opens a file, processes it according to the function, then closes it

12 Category Theory

This section will contain a brief look into category theory and how it applies to Haskell. Various concepts such as Applicatives and Monads are discussed mathematically; for a more haskell-focused approach, see chapter `Type Classes`.
A category is a collection of *objects* and *arrows*. Arrows acts as pathways between objects.

- $A \to B$ is a *morphism* from A to B.
- $A \to A$ is an *identity morphism* from A to B (endomorphism).
- Let $A \to B$ and $B \to C$ be denoted f and g, respectively. Then $A \to C$ is denoted as $g \circ f$ as a *composition*.

In a category, every object has at least one identity morphism, and every morphism is composable.
Notation:

- `obj(C)` := Class of objects in a category.
- `hom(C)` := Class of morphisms in a category.
- `C(a,b)` := All morphisms from a to b.
- $\circ$:= Composition of morphisms
  - $h \circ f \circ g \equiv (h \circ f) \circ g \equiv h \circ (f \circ g)$
  - $f \circ 1 \equiv 1 \circ f \equiv f$

In Haskell, types can be viewed as categories with functions acting as morphisms. Indeed, every function is composable, and the function `id` acts as an identity morphism.

```
import Control.Category

class Category (cat :: k -> k -> *) where
  id :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

12.1 Functors

A functor maps one category to another - in haskell, it maps some computation into the functorial context using `fmap`.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

The function `fmap` maps the morphism `a -> b` to `f a -> f b`.

12.2 Monoidal Category

Given a category C, a functor $\diamond$ (the "tensor product") where $\diamond : C \times C \to C$, and an identity element I with

- $\alpha_{A,B,C} := (A \diamond B) \diamond C \equiv A \diamond (B \diamond C)$
- $\lambda_A := I \diamond A \equiv A$
- $\rho_A := A \diamond I \equiv A$

A monoidal category is given by $(S, \{1\}, \diamond)$ where S is a set, 1 is the identity element, and $\diamond$ is a tensor product operation.

12.3 Monoidal Functors

Given two monoidal ctegories, $(C, 1_C, \diamond_C)$ and $(D, 1_D, \diamond_D)$ then $F : C \to D$ is a monoidal functor with

- $\phi_{A,B} := F(A) \diamond_D F(B) = F(A \diamond_C B)$
- $\phi := 1_D \to F(1_C)$

Let's define a monoidal functor in Haskell.

```haskell
class Functor f => Monoidal f where
  unit :: f ()
  (**) :: f a -> f b -> f (a, b)
```

This monoidal functor is different from a normal functor as `(**)` only works on objects which are in the same functorial context.
let's further define a function

```haskell
(<**>) :: Monoidal f => f (a -> b) -> f a -> f b
mf <**> mx = fmap (\(f,x) -> f x) (mf ** mx)

-- With this operator, we can lift any function into a functor
lift2 :: (a -> b -> c) -> (f a -> f b -> f c)
lift2 f x = (<**>) (fmap f x)

lift3 :: (a -> b -> c -> d) -> (f a -> f b -> f c -> f d)
lift3 f a b c = lift2 f a (b <**> c)

...

lift<n> f x1 ... xn = lift<n-1> f x1 ... x<n-1> <**> xn
```

12.4 Applicative Functors

Applicatives are simply equivalent to lax monoidal functors. These allow functions to be lifted into the functorial context in order to compose them.

```haskell
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

12.5 Monoids

Given a monoidal category $(C, 1, \diamond)$, then (M, μ, η) is a monoid iff

- M is an element in *obj* (C)
- $\mu : M \diamond M \to M$
- $\eta : 1 \to M$

12.6 Monads

A monad makes it possible to lift a value into the context and access with a function without loosing the context (important!).
Given a catgory C and a functor T with

- $T : C \to C$ (endofunctor)
- $\eta : 1_C \to T$
  Haskell: η `::` 1_C `-> T (a -> m a)`

- $\mu : T^2 \to T$

  Haskell: μ `::` T^2 `-> T (m (m a) -> m a)`

Such that

- $\mu \circ T\mu \equiv T\mu \circ \mu$
- $\mu \circ T\eta \equiv \mu \circ \eta T$

By these definitions, a useful property is that any number of applications of T can be reduces into a single application via μ.
A monad takes a category and puts it into a functorial context.
The following snippet illustrates η and μ definitions for `Maybe` as `unit` and `join` respectively,

```
unit :: a -> Maybe a
unit = Just

join :: Maybe (Maybe a) -> Maybe a
join (Just x) = x
join Nothing = Nothing
```

So far we have no functions to work with the values inside the Monad context.

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

We notice that the signature strongly resembles that of `fmap` for functors. Indeed, a sufficient definition is simply to re-use `fmap`.
Combined with the property of applicative functors to not leave their context, it makes sense in Haskell to define a Monad as a subclass of Applicative.

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

using the functions already defined, we can define these new functions

```
x >>= f = join (map f x)
a >> b = a >>= \_ -> b
return = unit
```

12.7 Arrows

Arrows is a structure representing the abstract concept of computation, spefically composition, parameterised by their input and output. Arrows are essentially functions lifted into a context.

```
import Control.Arrow -- Useful functions/classes found here!

class Category a => Arrow (a :: * -> * -> *) where
  arr :: (b -> c) -> a b c

  -- optional functions
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

- `arr` lifts a function into the arrow context. It takes a function `input -> output` and returns an Arrow instance with the same input and output; can be through of like `pure` for Arrows.
- `first` takes an existing arrow, and creates a new arrow which works on tuples. The function operates on the first argument of the tuple and preserves the second.

- `second` takes an existing arrow, and creates a new arrow which works on tuples. The function operates on the second argument of the tuple and preserves the first.
- `(***)` is a combination of both `first` and `second`, returning tuples which contain both the origin and transformed inputs.
- `(&&&)` transforms an argument in two different ways, returning both outputs.

12.7.1 Function Type

```
instance Arrow (->) where
  -- These are both necessary
  arr = id
  (***) f g (x, y) = (f x, f y)

  -- These are optional
  first f (x, y) = (f x, y)
  second f (x, y) = (x, f y)
  (&&&) f g (x, y) = (f x, g y)
```

12.7.2 Kleisli Arrows

```
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Arrow (Kleisli m) where
  arr f = Kleisli (return . f)
  first (Kleisli f) = Kleisli (\ ~(b,d) -> f b >>= \c -> return (c,d))
  second (Kleisli f) = Kleisli (\ ~(d,b) -> f b >>= \c -> return (d,c))
```

12.7.3 Choice Arrows

```
class Arrow a => ArrowChoice (a :: * -> * -> *) where
  left :: a b c -> a (Either b d) (Either c d) -- Only changes value of the Left constructor
  right :: a b c -> a (Either d b) (Either d c) -- Only changes value of the Right constructor
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b c -> a c d -> a (Either b c) d
```

12.7.4 Arrow Application

```
class Arrow a => ArrowApply (a : * -> * -> *) where
  app :: a (a b c, b) c

instance ArrowApply (->) where
  app (f, x) = f x
```

13 Type Families

Type families are functions at defined at the type level.
Example:

```
data Nat = Z | S Nat

-- value-level functions
add :: Nat -> Nat -> Nat
add Z b = b
add (S a) b = S (add a b)

-- Type-level type family
type family Add (a :: Nat) (b :: Nat) :: Nat where
  Add 'Z b = b
  Add ('S a) b = 'S (Add a b)
```

In the first line of the family definition, we list the types of the arguments and the rteurn type. We populate the block with function definitions. Generally, every equation defined up-front: these are called **closed** type families. We can also have **open** type families which do not define every equation.

```
data Bool = True | False

-- Type-level equivalent to match the value-level function
type family Not (b :: Bool) :: Bool where
  Not 'True = 'False
  Not 'False = 'True

not :: Bool b -> Bool (Not b)
not True = False
not False = True
```

The input type determines what the output type will be. The type family will be evaluated for each pattern match in our function equation.