# Haskell CheatSheet

## Hello, Home!

```
main = do putStr "What's your name? "
          name <- getLine
          putStrLn ("It's 2020, " ++ name ++ "! Stay home, stay safe!")
```

## Pattern Matching

Functions can be defined using the usual `if_then_else_` construct, or as expressions *guarded* by Boolean expressions as in mathematics, or by *pattern matching* —a form of 'syntactic comparision'.

```
fact n = if n == 0 then 1 else n * fact (n - 1)

fact' n | n == 0 = 1
        | n != 0 = n * fact' (n - 1)

fact'' 0 = 1
fact'' n = n * fact'' (n - 1)
```

The above definitions of the factorial function are all equal.

Guards, as in the second version, are a form of 'multi-branching conditional'.

In the final version, when a call, say, `fact 5` happens we compare *syntactically* whether `5` and the first pattern `0` are the same. They are not, so we consider the second case with the understanding that an identifier appearing in a pattern matches *any* argument, so the second clause is used.

Hence, when pattern matching is used, order of equations matters: If we declared the `n`-pattern first, then the call `fact 0` would match it and we end up with `0 * fact (-1)`, which is not what we want!

If we simply defined the final `fact` using *only* the first clause, then `fact 1` would crash with the error *Non-exhaustive patterns in function fact*. That is, we may define *partial functions* by not considering all possible shapes of inputs.

See also "view patterns".

## Local Bindings

An equation can be qualified by a `where` or `let` clause for defining values or functions used only within an expression.

```
  ...e...e...e where e = expr
≈ let e = expr in ...expr...expr...expr
```

It sometimes happens in functional programs that one clause of a function needs *part of* an argument, while another operators on the *whole* argument. It it tedious (and inefficient) to write out the structure of the complete argument again when referring to it. Use the "as operator" `@` to label all or part of an argument, as in

```
f label@(x:y:ys) = ···
```

## Operators

Infix operators in Haskell must consist entiry of 'symbols' such as `&, ^, !, ...` rather than alphanumeric characters. Hence, while addition, `+`, is written infix, integer division is written prefix with `div`.

We can always use whatever fixity we like:

- ◇ If `f` is any *prefix* binary function, then `x `f` y` is a valid *infix* call.
- ◇ If ⊕ is any *infix* binary operator, then `(⊕) x y` is a valid *prefix* call.

It is common to fix one argument ahead of time, e.g., `λ x → x + 1` is the successor operation and is written more tersely as `(+1)`. More generally, `(⊕r) = λ x → x ⊕ r`.

The usual arithmeic operations are `+, /, *, -` but `%` is used to make fractions.

The Boolean operations are `==, /=, &&, ||` for equality, discrepancy, conjunction, and disjunction.

## Types

Type are inferred, but it is better to write them explicitly so that *you communicate your intentions to the machine*. If you *think* that expression `e` has type $\tau$ then write `e ::` $\tau$ to *communicate* that to the machine, which will silently accept your claim or reject it loudly.

| Type | Name | Example Value |
|---|---|---|
| Small integers | `Int` | `42` |
| Unlimited integers | `Integer` | `7376541234` |
| Reals | `Float` | `3.14` and `2 % 5` |
| Booleans | `Boolean` | `True` and `False` |
| Characters | `Char` | `'a'` and `'3'` |
| Strings | `String` | `"salam"` |
| Lists | $[\alpha]$ | `[]` or $[x_1, \ldots, x_n]$ |
| Tuples | $(\alpha, \beta, \gamma)$ | $(x_1, x_2, x_3)$ |
| Functions | $\alpha \to \beta$ | $\lambda\ x \to \cdots$ |

*Polymorphism* is the concept that allows one function to operate on different types.

- ◇ A function whose type contains *variables* is called a *polymorphic function*.
- ◇ The simplest polymorphic function is `id :: a -> a`, defined by `id x = x`.

## Tuples

**Tuples** $(\alpha_1, \ldots, \alpha_n)$ are types with values written $(x_1, \ldots, x_n)$ where each $x_i$ :: $\alpha_i$. The are a form of 'record' or 'product' type.

E.g., `(True, 3, 'a') :: (Boolean, Int, Char)`.

Tuples are used to "return multiple values" from a function.

Two useful functions on tuples of length 2 are:

```
fst :: (α, β) → α
fst (x, y) = x

snd :: (α, β) → β
snd (x, y) = β
```

If in addition you `import Control.Arrow` then you may use:

```
first :: (α → τ) → (α, β) → (τ, β)
first f (x, y) = (f x, y)

second :: (β → τ) → (α, β) → (α, τ)
second g (x, y) = (x, g y)

(***) :: (α → α') → (β → β) → (α, β) → (α', β')
(f *** g) (x, y) = (f x, g y)

(&&&) :: (τ → α) → (τ → β) → τ → (α, β)
(f &&& g) x = (f x, g x)
```

## Lists

**Lists** are sequences of items of the same type.
If each $x_i$ :: $\alpha$ then $[x_1, \ldots, x_n]$ :: $[\alpha]$.

Lists are useful for functions that want to 'non-deterministicly' return a value: They return a list of all possible values.

- The *empty list* is `[]`
- We "cons"truct nonempty lists using `(:)` :: $\alpha \to [\alpha] \to [\alpha]$
- Abbreviation: $[x_1, \ldots, x_n] = x_1 : (x_2 : (\cdots (x_n : [])))$
- *List comprehensions*: `[f x | x <- xs, p x]` is the list of elements `f x` where `x` is an element from list `xs` and `x` satisfies the property `p`
  - E.g., `[2 * x | x <- [2, 3, 4], x < 4]` ≈ `[2 * 2, 2 * 3]` ≈ `[4, 6]`
- Shorthand notation for segments: `u` may be ommitted to yield *infinite lists*
  - `[1 .. u] = [1, 1 + 1, 1 + 2, ..., u]`.
  - `[a, b, .., u] = [a + i * step | i <- [0 .. u - a] ] where step = b - a`

**Strings** are just lists of characters: "$c_0c_1\ldots c_n$" ≈ ['$c_0$', ..., '$c_n$'].

- Hence, all list methods work for strings.

---

### Pattern matching on lists

```
prod []     = 1
prod (x:xs) = x * prod xs

fact n = prod [1 .. n]
```

If your function needs a case with a list of say, length 3, then you can match directly on that *shape* via `[x, y, z]` —which is just an abbreviation for the shape `x:y:z:[]`.

Likewise, if we want to consider lists of length *at least 3* then we match on the shape `x:y:z:zs`. E.g., define the function that produces the maximum of a non-empty list, or the function that removes adjacent duplicates —both require the use of guards.

---

```
[x0, ..., xn] !! i = xi
[x0, ..., xn] ++ [y0, ..., ym] = [x0, ..., xn, y0, ..., ym]
concat [xs0, ..., xsn] = xs0 ++ ··· ++ xsn

{- Partial functions -}
head [x0, ..., xn] = x0
tail [x0, ..., xn] = [x1, ..., xn]
init [x0, ..., xn] = [x0, ..., xn−1]
last [x0, ..., xn] = xn

take k [x0, ..., xn] = [x0, ..., xk−1]
drop k [x0, ..., xn] = [xk, ..., xn]

sum     [x0, ..., xn] =  x0 + ··· + xn
prod    [x0, ..., xn] =  x0 * ··· * xn
reverse [x0, ..., xn] =  [xn, ..., x0]
elem x  [x0, ..., xn] =  x == x0 || ··· || x == xn

zip [x0, ..., xn] [y0, ..., ym]  = [(x0, y0), ..., (xk, yk)] where k = n `min` m
unzip [(x0, y0), ..., (xk, yk)] = ([x0, ..., xk], [y0, ..., yk])
```

**Duality**: Let ∂`f = reverse . f . reverse`, then `init` = ∂ `tail` and `take k` = ∂ `(drop k)`; even `pure . head` = ∂ `(pure . last)` where `pure x = [x]`.

## List 'Design Patterns'

Many functions have the same 'form' or 'design pattern', a fact which is taken advantage of by defining *higher-order functions* to factor out the structural similarity of the individual functions.

---

`map f xs = [f x | x <- xs]`

- Transform all elements of a list according to the function `f`.

---

`filter p xs = [x | x <- xs, p x]`

- Keep only the elements of the list that satisfy the predicate `p`.
- `takeWhile p xs` ≈ Take elements of `xs` that satisfy `p`, but stop stop at the first element that does not satisfy `p`.
- `dropWhile p xs` ≈ Drop all elements until you see one that does not satisfy the predicate.
- `xs = takeWhile p xs ++ dropWhile p xs`.

---

Right-folds let us 'sum' up the elements of the list, associating to the right.

```
foldr (⊕) e ≈ λ (x0 : (x1 : (... : (xn : []))))
              → (x0 ⊕ (x1 ⊕ (... ⊕ (xn ⊕ e))))
```

This function just replaces cons "`:`" and `[]` with ⊕ and `e`. That's all.

- E.g., replacing `:,[]` with themselves does nothing: `foldr (:) [] = id`.

*All functions on lists can be written as folds!*

```
   h [] = e  ∧  h (x:xs) = x ⊕ h xs
≡  h = foldr (λ x rec_call → x ⊕ rec_call) e
```

- ◇ Look at the two cases of a function and move them to the two first arguments of the fold.
  - ○ `map f = foldr (λ x ys → f x :  ys) []`
  - ○ `filter p = foldr (λ x ys → if (p x) then (x:ys) else ys) []`
  - ○ `takeWhile p = foldr (λ x ys → if (p x) then (x:ys) else []) []`

You can also fold leftward, i.e., by associating to the left:

$$\texttt{foldl}\ (\oplus)\ \texttt{e} \quad \approx \quad \lambda\ \texttt{(x}_0\ \texttt{:}\ \texttt{(x}_1\ \texttt{:}\ \texttt{(...}\ \texttt{:}\ \texttt{(x}_n\ \texttt{:}\ \texttt{[])))) } \to \texttt{(((e}\ \oplus\ \texttt{x}_0\texttt{)}\ \oplus\ \texttt{x}_1\texttt{)}\ \oplus\ \texttt{...}\ \texttt{)}\ \oplus\ \texttt{x}_n$$

Unless the operation $\oplus$ is associative, the folds are generally different.

- ◇ E.g., `foldl (/) 1 [1..n]` $\approx$ `1 / n!` where `n !  = product [1..n]`.
- ◇ E.g., `-55 = foldl (-) 0 [1..10]` $\neq$ `foldr (-) 0 [1..10] = -5`.

If `h` swaps arguments —`h(x ⊕ y) = h y ⊕ h x`— then `h` swaps folds:
`h .  foldr (⊕) e = foldl (⊖) e'` where `e' = h e` and `x ⊖ y = x ⊕ h y`.

E.g., `foldl (-) 0 xs = - (foldr (+) 0 xs) = - (sum xs)`
and `n !  = foldr (*) 1 [1..n] = 1 / foldl (/) 1 [1..n]`.

*( Floating points are a leaky abstraction! )*

## Algebraic data types

When we have 'possible scenarios', we can make a type to consider each option. E.g., `data Door = Open | Closed` makes a new datatype with two different values. Under the hood, `Door` could be implemented as integers and `Open` is 0 and `Closed` is 1; or any other implementation —*all that matters* is that we have a new type, `Door`, with two different values, `Open` and `Closed`.

Usually, our scenarios contain a 'payload' of additional information; e.g., `data Door2 = Open | Ajar Int | Closed`. Here, we have a new way to construct `Door` values, such as `Ajar 10` and `Ajar 30`, that we could interpret as denoting how far the door is open/. Under the hood, `Door2` could be implemented as pairs of integers, with `Open` being `(0,0)`, `Ajar n` being `(1, n)`, and `Closed` being `(2, 0)` —i.e., as the pairs "(value position, payload data)". Unlike functions, a value construction such as `Ajar 10` cannot be simplified any further; just as the list value `1:2:3:[]` cannot be simplified any further. Remember, the representation under the hood does not matter, what matters is that we have three possible *construction forms* of `Door2` values.

Languages, such as C, which do not support such an "algebraic" approach, force you, the user, to actually choose a particular representation —even though, it does not matter, since we only want *a way to speak of* "different cases, with additional information".

In general, we declare the following to get an "enumerated type with payloads".

$$\texttt{data D = C}_0\ \tau_1\ \tau_2\ \texttt{...}\ \tau_m\ \texttt{|}\ \texttt{C}_1\ \cdots\ \texttt{|}\ \texttt{C}_n\ \cdots\ \texttt{deriving Show}$$

There are `n` constructors $\texttt{C}_i$ that make *different* values of type `D`; e.g., $\texttt{C}_0\ \texttt{x}_1\ \texttt{x}_2\ \texttt{...}\ \texttt{x}_m$ is a `D`-value whenever each $\texttt{x}_i$ is a $\tau_i$-value. The **''deriving Show''** at the end of the definition is necessary for user-defined types to make sure that values of these types can be printed in a standard form.

We may now define functions on `D` by pattern matching on the possible ways to *construct* values for it; i.e., by considering the cases $\texttt{C}_i$.

In-fact, we could have written `data D` $\alpha_1\ \alpha_2\ \ldots\ \alpha_k$ `= ⋯`, so that we speak of "D values *parameterised* by types $\alpha_i$". E.g., "lists whose elements are of type $\alpha$" is defined by `data List` $\alpha$ `= Nil | Cons` $\alpha$ `(List` $\alpha$`)` and, for example, `Cons 1 (Cons 2 Nil)` is a value of `List Int`, whereas `Cons 'a' Nil` is of type `List Char`. —The `List` type is missing the **''deriving Show''**, see below for how to *mixin* such a feature.

For example, suppose we want to distinguish whether we have an $\alpha$-value or a $\beta$-value, we use `Either`. Let's then define an example *infix* function using pattern matching.

```
data Either α β = Left α | Right β

(+++) :: (α → α') → (β → β') → Either α β → Either α' β'
(f +++ g) (Left  x) = Left $ f x
(f +++ g) (Right x) = Right $ g x

right :: (β → τ) → Either α β → Either α τ
right f = id +++ f
```

The above `(+++)` can be found in `Control.Arrow` and is also known as `either` in the standard library.

## Typeclasses and overloading

*Overloading* is using the same name to designate operations "of the same nature" on values of different types.

E.g., the `show` function converts its argument into a string; however, it is not polymorphic: We cannot define `show ::  α → String` with one definition since some items, like functions or infinite datatypes, cannot be printed and so this is not a valid type for the function `show`.

Haskell solves this by having `Show` *typeclass* whose *instance types* $\alpha$ each implement a definition of the *class method* `show`. The type of `show` is written `Show α => α -> String`: *Given an argument of type $\alpha$, look in the global listing of `Show` instances, find the one for $\alpha$, and use that;* if $\alpha$ has no `Show` instance, then we have a type error. One says "the type variable $\alpha$ has is *restricted* to be a `Show` instance" —as indicated on the left side of the **''=>''** symbol.

E.g., for the `List` datatype we defined, we may declare it to be 'showable' like so:

```
1  instance Show a => Show (List a) where
2    show Nil         = "Nope, nothing here"
3    show (Cons x xs) = "Saw " ++ show x ++ ", then " ++ show xs
```

That is:

1. *If `a` is showable, then `List a` is also showable.*
2. *Here's how to show `Nil` directly.*
3. *We show `Cons x xs` by using the `show` of `a` on `x`, then recursively showing `xs`.*

| Common Typeclasses | |
|---|---|
| `Show` | Show elements as strings, `show` |
| `Read` | How to read element values from strings, `read` |
| `Eq` | Compare elements for equality, `==` |
| `Num` | Use literals `0, 20, ...`, and arithmetic `+, *, -` |
| `Ord` | Use comparison relations `>, <, >=, <=` |
| `Enum` | Types that can be listed, `[start .. end]` |
| `Monoid` | Types that model '(untyped) composition' |
| `Functor` | *Type formers* that model effectful computation |
| `Applicative` | Type formers that can sequence effects |
| `Monad` | Type formers that let effects depend on each other |

The `Ord` typeclass is declared `class Eq a => Ord a where ···`, so that all ordered types are necessarily also types with equality. One says `Ord` is a *subclass* of `Eq`; and since subclasses *inherit* all functions of a class, we may always replace `(Eq a, Ord a) => ···` by `Ord a => ···`.

You can of-course define your own typeclasses; e.g., the `Monoid` class in Haskell could be defined as follows.

```
class Semigroup a where
  (<>) :: a -> a -> a  {- A way to "compose" elements together -}
  {- Axiom: (x <> y) <> z = x <> (y <> z) -}

class Semigroup a => Monoid a where
  mempty :: a   {- Axiom: This is a 'no-op', identity, for composition <> -}
```

Example monoids `(α, <>, mempty)` include `(Int, +, 0)`, `([α], ++, [])`, and (Program statements, sequence ";", the empty statement) —this last example is approximated as `Term` with 'let-in' clauses at the end of this cheatsheet. *Typeclasses are interfaces, possibly with axioms specifying their behaviour.*

As shown earlier, Haskell provides a the `deriving` mechanism for making it easier to define instances of typeclasses, such as `Show, Read, Eq, Ord, Enum`. How? Constructor names are printed and read as written as written in the `data` declaration, two values are equal if they are formed by the same construction, one value is less than another if the constructor of the first is declared in the `data` definition before the constructor of the second, and similarly for listing elements out.

## Functor

*Functors are type formers that "behave" like collections: We can alter their "elements" without messing with the 'collection structure' or 'element positions'.* The well-behavedness constraints are called *the functor axioms.*

```
class Functor f where
  fmap :: (α → β) → f α → f β

(<$>) = fmap {- An infix alias -}
```

The axioms cannot be checked by Haskell, so we can form instances that fail to meet the implicit specifications —two examples are below.

---

**Identity Law**: `fmap id = id`

*Doing no alteration to the contents of a collection does nothing to the collection.*

This ensures that "alterations don't needlessly mess with element values" e.g., the following is not a functor since it does.

```
{- I probably have an item -}
data Probably a = Chance a Int

instance Functor Probably where
  fmap f (Chance x n) = Chance (f x) (n `div` 2)
```

---

**Fusion Law:** `fmap f . fmap g = fmap (f . g)`

*Reaching into a collection and altering twice is the same as reaching in and altering once.*

This ensures that "alterations don't needlessly mess with collection structure"; e.g., the following is not a functor since it does.

```
import Prelude hiding (Left, Right)

{- I have an item in my left or my right pocket -}
data Pocket a = Left a | Right a

instance Functor Pocket where
  fmap f (Left  x) = Right (f x)
  fmap f (Right x) = Left  (f x)
```

---

It is important to note that functors model well-behaved container-like types, but of-course the types do not actually need to contain anything at all! E.g., the following is a valid functor.

```
{- "I totally have an α-value, it's either here or there." Lies! -}
data Liar α = OverHere Int | OverThere Int

instance Functor Liar where
  fmap f (OverHere  n) = OverHere  n
  fmap f (OverThere n) = OverThere n
```

Notice that if we altered `n`, say by dividing it by two, then we break the identity law; and if we swap the constructors, then we break the fusion law. Super neat stuff!

In general, functors take something boring and generally furnish it with 'coherent' structure, but **there is not necessarily an** $\alpha$ **'inside'** `f` $\alpha$. E.g., `f α = (ϵ → α)` has as values "recipes for forming an $\alpha$-value", but unless executed, there is no $\alpha$-value.

---

- `fmap f xs` ≈ *for each* element `x` in the 'collection' `xs`, yield `f x`.
- Haskell can usually `derive` functor instances since they are unique: Only one possible definition of `fmap` will work.
- Reading the functor axioms left-to-right, they can be seen as *optimisation laws* that make a program faster by reducing work.
- The two laws together say *fmap distributes over composition*: `fmap (f1 . f2 . ··· . fn) = fmap f1 . ··· . fmap fn` for `n` $\geq 0$.

**Naturality Theorems:** If `p :: f a → g a` for some *functors* `f` and `g`, then `fmap f . p = p . fmap f` for any *function* `f`.

Hence, any generic property `p :: f α → ϵ` is invariant over fmaps: `p(fmap f xs) = p xs`. E.g., the length of a list does not change even when an fmap is applied.

## Functor Examples

Let $\texttt{f}_1$, $\texttt{f}_2$ be functors and $\epsilon$ be a given type.

| Type Former | `f` $\alpha$ | `f <$> x` |
|---|---|---|
| Identity | $\alpha$ | `f <$> x = f x` |
| Constant | $\epsilon$ | `f <$> x = x` |
| List | $[\alpha]$ | `f <$> [x0, ..., xn] = [f x0, ..., f xn]` |
| Either | `Either` $\epsilon$ $\alpha$ | `f <$> x = right f` |
| Product | $(\texttt{f}_1\ \alpha, \texttt{f}_2\ \alpha)$ | `f <$> (x, y) = (f <$> x, f <$> y)` |
| Composition | $\texttt{f}_1\ (\texttt{f}_2\ \alpha)$ | `f <$> x = (fmap f) <$> x` |
| Sum | `Either` $(\texttt{f}_1\ \alpha)\ (\texttt{f}_2\ \alpha)$ | `f <$> ea = f +++ f` |
| Writer | $(\epsilon, \alpha)$ | `f <$> (e, x) = (e, f x)` |
| Reader | $\epsilon \to \alpha$ | `f <$> g = f . g` |
| State | $\epsilon \to (\epsilon, \alpha)$ | `f <$> g = second f . g` |

Notice that writer is the product of the constant and the identity functors.

Unlike reader, the type former `f` $\alpha$ `=` $\alpha \to \epsilon$ is *not* a functor since there is no way to implement `fmap`. In contrast, `f` $\alpha$ `=` $(\alpha \to \epsilon, \alpha)$ *does* have an implementation of `fmap`, but it is not lawful.

## Applicative

*Applicatives are collection-like types that can apply collections of functions to collections of elements.*

In particular, *applicatives can fmap over multiple arguments*; e.g., if we try to add `Just 2` and `Just 3`, we find `(+) <$> Just 2 :: Maybe (Int → Int)` and this is not a function and so cannot be applied further to `Just 3` to get `Just 5`. We have both the function and the value wrapped up, so we need a way to apply the former to the latter. The answer is `(+) <$> Just 2 <*> Just 3`.

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

{- Apply associates to the left: p <*> q <*> r = (p <*> q) <*> r) -}
```

The method `pure` lets us inject values, to make 'singleton collections'.

- ◇ *Functors transform values inside collections; applicatives can additionally combine values inside collections.*
- ◇ Exercise: If $\alpha$ is a monoid, then so too is `f` $\alpha$ for any applicative `f`.

---

The applicative axioms ensure that apply behaves like usual functional application:

- ◇ Identity: `pure id <*> x = x` —c.f., `id x = x`
- ◇ Homomorphism: `pure f <*> pure x = pure (f x)` —it really is function application on pure values!
  - ○ Applying a non-effectful function to a non-effectful argument in an effectful context is the same as just applying the function to the argument and then injecting the result into the content.
- ◇ Interchange: `p <*> pure x = pure ($ x) <*> p` —c.f., `f x = ($ x) f`
  - ○ Functions `f` take `x` as input ≈ Values `x` project functions `f` to particular values
  - ○ When there is only one effectful component, then it does not matter whether we evaluate the function first or the argument first, there will still only be one effect.
  - ○ Indeed, this is equivalent to the law: `pure f <*> q = pure (flip ($)) <*> q <*> pure f`.
- ◇ Composition: `pure (.) <*> p <*> q <*> r = p <*> (q <*> r)` —c.f., `(f . g) . h = f . (g . h)`.

If we view `f` $\alpha$ as an "effectful computation on $\alpha$", then the above laws ensure `pure` creates an "effect free" context. E.g., if `f` $\alpha$ `=` $[\alpha]$ is considered "nondeterminstic $\alpha$-values", then `pure` just treats usual $\alpha$-values as nondeterminstic but with no ambiguity, and `fs <*> xs` reads "if we nondeterminsticly have a choice `f` from `fs`, and we nondeterminsticly an `x` from `xs`, then we nondeterminsticly obtain `f x`." More concretely, if I'm given randomly addition or multiplication along with the argument 3 and another argument that could be 2, 4, or 6, then the result would be obtained by considering all possible combinations: `[(+), (*)] <*> pure 3 <*> [2, 4, 6] = [5,7,9,6,12,18]`. The name "`<*>`" is suggestive of this 'cartesian product' nature.

Given a definition of apply, the definition of `pure` may be obtained by unfolding the identity axiom.

Using these laws, we regain the original `fmap` —since `fmap`'s are unique in Haskell— thereby further cementing that applicatives model "collections that can be functionally applied": `f <$> x = pure f <*> x`. ( Hence, every applicative is a functor whether we like it or not. )

- ◇ The identity applicative law is then just the identity law of functor.
- ◇ The homomorphism law now becomes: `pure . f = fmap f . pure`.
  - ○ This is the "naturality law" for `pure`.

---

The laws may be interpreted as left-to-right rewrite rules and so are a procedure for transforming any applicative expression into the canonical form of "a pure function applied to effectful arguments": `pure f <*> x1 <*> ⋯ <*> xn`. In this way, one can compute in-parallel the, necessarily independent, $\texttt{x}_i$ then combine them together.

Notice that the canonical form generalises `fmap` to `n`-arguments:
Given `f` $:: \alpha_1 \to \cdots \to \alpha_n \to \beta$ and $\texttt{x}_i$ `::` `f` $\alpha_i$, we obtain an `(f` $\beta$`)`-value.
The case of `n = 2` is called `liftA2`, `n = 1` is just `fmap`, and for `n = 0` we have `pure`!

---

Notice that `lift2A` is essentially the cartesian product in the setting of lists, or `(<&>)` below —c.f., `sequenceA :: Applicative f ⇒ [f a] → f [a]`.

```
(<&>) :: f a → f b → f (a, b)   {- Not a standard name! -}
(<&>) = liftA2 (,)  -- i.e., p <&> q = (,) <$> p <*> q
```

This is a pairing operation with properties of `(,)` mirrored at the applicative level:

```
{- Pure Pairing -} pure x <&> pure y = pure (x, y)
{- Naturality   -} (f &&& g) <$> (u <&> v) = (f <$> u) <&> (g <&> v)

{- Left Projection  -} fst <$> (u <&> pure ()) = u
{- Right Projection -} snd <$> (pure () <&> v)  = v
{- Associtivity     -} assocl <$> (u <&> (v <&> w)) = (u <&> v) <&> w
```

The final three laws above suffice to prove the original applicative axioms, and so we may define `p <*> q = uncurry ($) <$> (p <&> q)`.

## Applicative Examples

Let `f1`, `f2` be functors and let $\epsilon$ a type.

| Functor | `f` $\alpha$ | `f <*> x` |
|---|---|---|
| Identity | $\alpha$ | `f <*> x = f x` |
| Constant | $\epsilon$ | `e <*> d = e <> d` |
| List | $[\alpha]$ | `fs <*> xs = [f x \| f <- fs, x <- xs]` |
| Either | `Either` $\epsilon$ $\alpha$ | `ef <*> ea = right (λ f → right f ea) ef` |
| Composition | `f1 (f2` $\alpha$`)` | `f <*> x = (<*>) <$> f <*> x` |
| Product | `(f1` $\alpha$`, f2` $\alpha$`)` | `(f, g) <*> (x, y) = (f <*> x, g <*> y)` |
| Sum | `Either (f1` $\alpha$`) (f2` $\alpha$`)` | Challenge: Assume $\eta$ `:: f1 a →  f2 a` |
| Writer | $(\epsilon, \alpha)$ | `(a , f) <*> (b, x) = (a <> b, f x)` |
| Reader | $\epsilon \to \alpha$ | `f <*> g = λ e → f e (g e)` —c.f., `SKI` |
| State | $\epsilon \to (\epsilon, \alpha)$ | `sf <*> sa = λ e → let (e', f) = sf e in second f (sa e')` |

In the writer and constant cases, we need $\epsilon$ to also be a monoid. When $\epsilon$ is *not* a monoid, then those two constructions give examples of functors that are *not* applicatives —since there is no way to define `pure`. In contrast, `f` $\alpha$ `= (`$\alpha \to \epsilon$`) → Maybe` $\epsilon$ is not an applicative since no definition of apply is lawful.

Since readers `((->) r)` are applicatives, we may, for example, write $(\oplus)$ `<$> f <*> g` as a terse alternative to the "pointwise $\oplus$" method $\lambda$ `x →  f x` $\oplus$ `g x`. E.g., using `(&&)` gives a simple way to chain predicates.

## Do-Notation —Subtle difference between applicatives and monads

Recall the `map` operation on lists, we could define it ourselves:

```
map' :: (α -> β) -> [α] -> [β]
map' f []     = []
map' f (x:xs) = let y  = f x
                    ys = map' f xs
                in  (y:ys)
```

If instead the altering function `f` returned effectful results, then we could gather the results along with the effect:

```
{-# LANGUAGE ApplicativeDo #-}

mapA :: Applicative f => (a -> f b) -> [a] -> f [b]
mapA f []     = pure []
mapA f (x:xs) = do y  <- f x
                   ys <- mapA f xs
                   pure (y:ys)
                {- ≈ (:) <$> f x <*> mapA f xs -}
```

Applicative syntax can be a bit hard to write, whereas `do`-notation is more natural and reminiscent of the imperative style used in defining `map'` above. For instance, the intuition that `fs <*> ps` is a cartesian product is clearer in do-notation: `fs <*> ps` ≈ `do {f ← fs; x ← ps; pure (f x)}` where the right side is read *"for-each f in fs, and each x in ps, compute f x"*.

In-general, `do {x1 ← p1; ...; xn ← pn; pure e}` ≈ `pure (λ x1 ... xn → e) <*> p1 <*> ··· <*> pn` **provided** `pi` does not mention `xj` for `j < i`; but `e` may refer to all `xi`. If any `pi` mentions an earlier `xj`, then we could not translate the `do`-notation into an applicative expression.

If `do {x ← p; y ← qx; pure e}` has `qx` being an expression **depending** on `x`, then we could say this is an abbreviation for `(λ x → (λ y → e) <$> qx) <$> p` but this is of type `f (f` $\beta$`))`. Hence, to allow later computations to depend on earlier computations, we need a method `join ::  f (f` $\alpha$`) → f` $\alpha$ with which we define `do {x ← p; y ← qx; pure e}` ≈ `join $ ~(λ x -> (λ y → e) <$> qx) <$> p`.

Applicatives with a `join` are called monads and they give us a **"programmable semicolon"**. Since later items may depend on earlier ones, `do {x ← p; y ← q; pure e}` could be read *"let x be the value of computation p, let y be the value of computation q, then combine the values via expression e"*. Depending on how `<*>` is implemented, such 'let declarations' could short-circuit (`Maybe`) or be nondeterministic (`List`) or have other effects such as altering state.

As the `do`-notation clearly shows, the primary difference between `Monad` and `Applicative` is that `Monad` allows dependencies on previous results, whereas `Applicative` does not.

---

Do-syntax also works with tuples and functions –c.f., reader monad below— since they are monadic; e.g., every clause `x <- f` in a functional do-expression denotes the resulting of applying `f` to the (implicit) input. More concretely:

```
go :: (Show a, Num a) => a -> (a, String)
go = do {x <- (1+); y <- show; return (x, y)}

-- go 3 = (4, "3")
```

Likewise, tuples, lists, etc.

## Formal Definition of Do-Notation

For a general applicative `f`, a `do` expression has the form `do {C; r}`, where `C` is a (possibly empty) list of commands separated by semicolons, and `r` is an expression of type `f` $\beta$, which is also the type of the entire `do` expression. Each command takes the form `x ← p`, where `x` is a variable, or possibly a pattern; if `p ::  f` $\alpha$ then `x ::  `$\alpha$. In the particular case of the anonymous variable, `_ ← p` may be abbreviated to `p`.

The translation of a `do` expression into `<*>`/`join` operations and `where` clauses is governed by three rules —the last one only applies in the setting of a monad.

```
(1)  do {r}            = r
(2A) do {x ← p; C; r} = q <*> p where q x = do {C; r} --Provided x ∉ C
(2M) do {x ← p; C; r} = join $ map q p where q x = do {C; r}

{- Fact: When x ∉ C, (2A) = (2M). -}
```

By definition chasing and induction on the number of commands `C`, we have:

```
[CollapseLaw]  do {C; do {D; r}} = do {C; D; r}
```

Likewise:

```
[Map ] fmap f p = do {x ← p; pure (f x)} -- By applicative laws
[Join] join ps  = do {p ← ps; p}         -- By functor laws
```

**Do-Notation Laws**: Here are some desirable usability properties of `do`-notation.

```
[RightIdentity]  do {B; x ← p; pure x}       = do {B; p}
[LeftIdentity ]  do {B; x ← pure e; C; r}    = do {B; C[x ≔ e]; r[x ≔ e]}
[Associtivity ]  do {B; x ← do {C; p}; D; r} = do {B; C; x ← p; D; r}
```

Here, `B, C, D` range over sequences of commands and `C[x ≔ e]` means the sequence `C` with all free occruences of `x` replaced by `e`.

- ◇ Associtivity gives us a nice way to 'inline' other calls.
- ◇ The LeftIdentity law, read right-to-left, lets us "locally give a name" to the possibly complex expression `e`.

  If `pure` forms a singleton collection, then LeftIdentity is a "one-point rule": We consider *all* `x ← pure e`, but there is only *one* such `x`, namely `e`!

In the applicative case, where the clauses are independent, we can prove, say, `RightIdentity` using the identity law for applicatives —which says essentially `do {x <- p; pure x} = p`— then apply induction on the length of `B`.

What axioms are needed for the monad case to prove the `do`-notation laws?

## Monad Laws

Here is the definition of the monad typeclass.

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

(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
f <=< g = join . fmap f . g
```

Where's `join`!? Historically, monads entered Haskell first with interface `(>>=), return`; later it was realised that `return = pure` and the relationship with applicative was cemented.

'Bind' `(>>=)` is definable from `join` by `ma >>= f = join (fmap f ma)`, and, for this reason, bind is known as "flat map" or "concat map" in particular instances. For instance, the second definition of `do`-notation could be expressed:

```
(2M') do {x ← p; C; r} = p >>= q where q x = do {C; r}
```

Conversely, `join ps = do {p ← ps; p} = ps >>= id`. Likewise, with (2M'), note how `(<*>)` can be defined directly in-terms of `(>>=)`
—c.f., `mf <*> mx = do {f ← mf; x ← mx; return (f x)}`.

Since `fmap f p = do {x ← p; return (f x)} = p >>= return . f`, in the past monad did not even have functor as a superclass —c.f., liftM.

The properties of `>>=, return` that prove the desired `do`-notation laws are:

```
[LeftIdentity ] return a >>= f   ≡  f a
[RightIdentity] m >>= return     ≡  m
[Associtivity ] (m >>= f) >>= g  ≡  m >>= (\x -> f x >>= g)
                i.e.,   (m  >>=  (\x -> f x))  >>=  g
                      = m  >>=  (\x -> f x    >>=  g)
```

Equivalently, show the 'fish' `(<=<)` is associative with identity being `pure` —c.f., monoids!

It is pretty awesome that `(>>=), return` give us a functor, an applicative, and (dependent) do-notation! Why? Because bind does both the work of `fmap` and `join`. Thus, `pure, fmap, join` suffice to characterise a monad.

*Join determines how a monad behaves!*

The monad laws can be expressed in terms of `join` directly:

```
[Associativity] join . fmap join = join . join
{- The only two ways to get from "m (m (m α))" to "m α" are the same. -}

[Identity Laws] join. fmap pure = join . pure = id
{- Wrapping up "m α" gives an "m (m α)" which flattens to the original element. -}
```

Then, notice that the (free) naturality of join is:

```
join . fmap (fmap f) = fmap f . join  ::  m (m α) → m β
```

Again, note that `join` doesn't merely flatten a monad value, but rather performs the necessary logic that determines *how the monad behaves*.

E.g., suppose `m α = ϵ → (ϵ, α)` is the type of $\alpha$-values that can be configured according to a fixed environment type $\epsilon$, along with the possibly updated configuration —i.e., functions `ϵ → (ϵ, α)`. Then any `a : ϵ → (ϵ, ϵ → (ϵ, α))` in `m (m α)` can be considered an element of `m α` if we *propagate the environment configuration* through the outer layer to obtain a new configuration for the inner layer: `λ e → let (e', a') = a e in a' e'`. The join dictates how a configuration is *modified then passed along*: We have two actions, `a` and `a'`, and join has *sequenced* them by pushing the environment through the first thereby modifying it then pushing it through the second.

## Monad Examples

Let $\mathtt{f}_1$, $\mathtt{f}_2$ be functors and let $\epsilon$ a type.

| Applicative | `m α` | `join :: m (m α) → m α` |
|---|---|---|
| Identity | $\alpha$ | `λ x → x` |
| Constant | $\epsilon$ | `λ x → x` —Shucks! Not a monad! |
| List | `[α]` | `λ xss → foldr (++) [] xss` |
| Either | `Either ϵ α` | Exercise ^_^ |
| Composition | $\mathtt{f}_1\ (\mathtt{f}_2\ \alpha)$ | Nope! Not a monad! |
| Product | $(\mathtt{f}_1\ \alpha, \mathtt{f}_2\ \alpha)$ | `λ p → (fst <$> p, snd <$> p)` |
| Writer | `(ϵ, α)` | `λ (e, (e', a)) → (e <> e', a)` |
| Reader | `ϵ → α` | `λ ra → λ e → ra e e` |
| State | `ϵ → (ϵ, α)` | `λ ra → λ e → let (e', a) = ra e in a e'` |

In writer, we need $\epsilon$ to be a monoid.

- ◇ Notice how, in writer, join merges the outer context with the inner context: *Sequential writes are mappended together!*
- ◇ If `pure` forms 'singleton containers' then `join` flattens containers of containers into a single container.

Excluding the trivial monoid, the constant functor is *not* a monad: It fails the monad identity laws for join. Similarly, `f α = Maybe (α, α)` is an applicative but *not* a monad —since there is no lawful definition of `join`. Hence, applicatives are strictly more generally than monads.

## Running Example —A Simple Arithmetic Language

Let's start with a weak language:

```
data Term = Int Int | Div Term Term deriving Show

thirteen = Int 1729 `Div` (Int 133 `Div` Int 1)
boom     = Int 1729 `Div` (Int 12  `Div` Int 0)

eval₀ :: Term -> Int
eval₀ (Int n) =  n
eval₀ (n `Div` d) = let top    = eval₀ n
                        bottom = eval₀ d
                    in  top `div` bottom
```

How do we accomodate safe division by zero? Print to the user what's happening at each step of the calcuation? Have terms that access 'global' variables? Have terms that can store named expressions then access them later?

We'll make such languages and their `eval`'s will be nearly just as simple as this one (!) but accomodate these other issues.

## Maybe —Possibly Failing Computations

Safe evaluator: No division errors.

```
eval₁ :: Term -> Maybe Int
eval₁ (Int n) = pure n
eval₁ (n `Div` d) = do t <- eval₁ n
                       b <- eval₁ d
                       if b == 0 then Nothing else pure (t `div` b)
```

Exercise: Rewrite $\texttt{eval}_i$ without `do`-notation and you'll end-up with nested case analysis leading into a straicase of code that runs right off the page.

- ◇ Applicative is enough for $\texttt{eval}_1$, $\texttt{eval}_2$, $\texttt{eval}_3$, but $\texttt{eval}_4$ needs `Monad`.

## Writer —Logging Information as we Compute

Use a pair type `W` $\epsilon$ $\alpha$ to keep track of an environment $\epsilon$ and a value $\alpha$.

```
data Writer ϵ α = W ϵ α deriving Show

write :: ϵ -> Writer ϵ ()
write e = W e ()

instance Functor (Writer ϵ) where
  fmap f (W e a) = W e (f a)
```

Aggregate, merge, environments using their monoidal operation.

```
instance Monoid ϵ => Applicative (Writer ϵ) where
  pure a = W mempty a
  (W e f) <*> (W d a) = W (e <> d) (f a)

instance Monoid ϵ => Monad (Writer ϵ) where
  (>>=) = \ ma f -> join (pure f <*> ma)
    where join (W e (W d a)) = W (e <> d) a
```

An evaluator that prints to the user what's going on.

```
eval₂ :: Term -> Writer String Int
eval₂ it@(Int n) = W ("\n Evaluating: " ++ show it) n
eval₂ it@(n `Div` d) = do write $ "\n Evaluating: " ++ show it
                          t <- eval₂ n
                          b <- eval₂ d
                          pure $ (t `div` b)

-- Try this! With "boom", we get to see up to the boint of the error ^_^
-- let W e x = eval₂ thirteen in putStrLn e
```

## Reader —Accessing 'Global, read-only, data'

Use a function type $\epsilon \to \alpha$ to get $\alpha$-values that 'reads' from a configuration environment $\epsilon$.

```
data Reader ϵ α = R {run :: ϵ -> α}

instance Functor (Reader ϵ) where
  fmap f (R g) = R $ f . g

instance Applicative (Reader ϵ) where
  pure a = R $ const a
  (R f) <*> (R g) = R $ \e -> f e (g e) {- "S" combinator -}

instance Monad (Reader ϵ) where
  ma >>= f = join (pure f <*> ma)
    where join (R rf) = R $ \e -> run (rf e) e
```

A language with access to global variables; uninitialised variables are 0 by default.

```
data Term = Int Int | Div Term Term | Var String deriving Show

type GlobalVars = [(String, Int)]

valuefrom :: String -> GlobalVars -> Int
valuefrom x gvs = maybe 0 id $ lookup x gvs

eval₃ :: Term -> Reader GlobalVars Int
eval₃ (Int x) = pure x
eval₃ (Var x) = R $ \e -> x `valuefrom` e
eval₃ (n `Div` d) = do t <- eval₃ n
                       b <- eval₃ d
                       pure (t `div` b)

state    = [("x", 1729), ("y", 133)] :: GlobalVars
thirteen = Var "x" `Div` (Var "y" `Div` Int 1)
-- run (eval₃ thirteen) state
```

## State —Read and write to local storage

Let's combine writer and reader to get state: We can both read and write to data by using functions $\epsilon \to (\epsilon, \alpha)$ that read from an environment $\epsilon$ and result in a new environment as well as a value.

- `IO` $\alpha \cong$ `State TheRealWorld` $\alpha$ ;-)

```
data State ϵ α = S {run :: ϵ -> (ϵ, α)}

push :: Monoid ϵ => ϵ -> State ϵ ()
push d = S $ \e -> (d <> e, ())

instance Functor (State ϵ) where
  fmap f (S g) = S $ \ e -> let (e', a) = g e in (e', f a)

instance Applicative (State ϵ) where
  pure a = S $ \e -> (e, a)
  (S sf) <*> (S g) = S $ \e -> let (e',  a) = g e
                                   (e'', f) = sf e' in (e'', f a)

instance Monad (State ϵ) where
  ma >>= f = join (pure f <*> ma)
    where join (S sf) = S $ \e -> let (e', S f) = sf e in f e'
```

A simple language with storage; a program's value is the value of its final store.

```
data Expr = Let String Expr Expr | Var String | Int Int | Div Expr Expr
          deriving Show

eval₄ :: Expr -> State GlobalVars Int
eval₄ (Var x) = S $ \e -> let r = x `valuefrom` e in ((x,r):e, r)
eval₄ (Int x) = pure x
eval₄ (Let x t body) = do n <- eval₄ t
                          push [(x, n)] -- Applicative is NOT enough here!
                          eval₄ body
eval₄ (n `Div` d) = do t <- eval₄ n; b <- eval₄ d; pure (t `div` b)

thirteen = Let "x" (Int 1729)
             $ Let "y" (Int 133 `Div` Int 1)
                $ Var "x" `Div` Var "y"

-- run (eval₄ thirteen) []
```

Exercise: Add to the oringal `Term` type a constructor `Rndm [Term]`, where `Rndm [`$t_1$`, ..., `$t_n$`]` denotes non-deterministicly choosing one of the terms $t_i$. Then write an evaluator that considers all possible branches of a computation: `eval`$_5$ `: Term` $\to$ `[Int]`.

If we want to mixin any of the features for our evaluators, we need to use 'monad transformers' since monads do not compose in general.

## Reads

- *Introduction to Functional Programming* by Richard Bird
  - Assuming no programming, this book end by showing how to write a theorem prover powerful enough to prove many of laws scattered throughout the book.
- Monads for functional programming by Philip Wadler
  - This covers the `eval`$_i$ and more ^_^
- Comprehending Monads by Philip Wadler
- What I Wish I Knew When Learning Haskell
- Typeclassopedia —*The essentials of each type class are introduced, with examples, commentary, and extensive references for further reading.*
- You Could Have Invented Monads! (And Maybe You Already Have.)
- Learn You a Haskell for Great Good —An accessible read with many examples, and drawings
- The Haskell WikiBook —Has four beginner's tracks and four advanced tracks
- Category Theory Cheat Sheet —The "theory of typed composition": Products, Sums, Functors, Natural Transformations ^_^
- Agda Cheat Sheet —Agda is Haskell on steroids in that it you can invoke Haskell code and write proofs for it.
- LINQ for applicatives and monads.
  - Monads ≈ SQL/Linq ≈ Comprehensions/Generators