

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 comparison’.

```
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 is 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 entirely 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., $\lambda x \rightarrow x + 1$ is the successor operation and is written more tersely as `(+1)`. More generally, $(\oplus r) = \lambda x \rightarrow x \oplus r$.

The usual arithmetic 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 τ 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 $[x_1, \dots, x_n]$
Tuples	(α, β, γ)	(x_1, x_2, x_3)
Functions	$\alpha \rightarrow \beta$	$\lambda x \rightarrow \dots$

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 $\text{id} :: a \rightarrow a$, defined by $\text{id } x = x$.

Tuples

Tuples $(\alpha_1, \dots, \alpha_n)$ are types with values written (x_1, \dots, x_n) where each $x_i :: \alpha_i$. They are a form of ‘record’ or ‘product’ type.

E.g., $(\text{True}, 3, \text{'a'}) :: (\text{Boolean}, \text{Int}, \text{Char})$.

Tuples are used to “return multiple values” from a function.

Two useful functions on tuples of length 2 are:

```
fst :: ( $\alpha, \beta$ )  $\rightarrow \alpha$ 
fst (x, y) = x
```

```
snd :: ( $\alpha, \beta$ )  $\rightarrow \beta$ 
snd (x, y) = y
```

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

```
first :: ( $\alpha \rightarrow \tau$ )  $\rightarrow (\alpha, \beta) \rightarrow (\tau, \beta)$ 
first f (x, y) = (f x, y)
```

```
second :: ( $\beta \rightarrow \tau$ )  $\rightarrow (\alpha, \beta) \rightarrow (\alpha, \tau)$ 
second g (x, y) = (x, g y)
```

```
(***) :: ( $\alpha \rightarrow \alpha'$ )  $\rightarrow (\beta \rightarrow \beta') \rightarrow (\alpha, \beta) \rightarrow (\alpha', \beta')$ 
(f ***) g (x, y) = (f x, g y)
```

```
(&&&) :: ( $\tau \rightarrow \alpha$ )  $\rightarrow (\tau \rightarrow \beta) \rightarrow \tau \rightarrow (\alpha, \beta)$ 
(f &&& g) x = (f x, g x)
```

Lists

Lists are sequences of items of the same type.

If each $x_i :: \alpha$ then $[x_1, \dots, x_n] :: [\alpha]$.

- ◊ The *empty list* is []
- ◊ We “construct” nonempty lists using $(:)$ $:: \alpha \rightarrow [\alpha] \rightarrow [\alpha]$
- ◊ Abbreviation: $[x_1, \dots, x_n] = x_1 : (x_2 : (\dots (x_n : [])))$
- ◊ *List comprehensions*: $[f x \mid x \leftarrow xs, p x]$ is the list of elements $f x$ where x is an element from list xs and x satisfies the property p
 - ◊ E.g., $[2 * x \mid x \leftarrow [2, 3, 4], x < 4] \approx [2 * 2, 2 * 3] \approx [4, 6]$
- ◊ Shorthand notation for segments: u may be omitted to yield *infinite lists*
 - ◊ $[1 .. u] = [1, 1 + 1, 1 + 2, \dots, u]$.
 - ◊ $[a, b, .., u] = [a + i * \text{step} \mid i \leftarrow [0 .. u - a]]$ where $\text{step} = b - a$

Strings are just lists of characters: $"c_0c_1\dots c_n" \approx [\text{'c}_0', \dots, \text{'c}_n']$.

- ◊ 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 $\partial f = \text{reverse} \circ f \circ \text{reverse}$, then $\text{init} = \partial \text{tail}$ and $\text{take } k = \partial (\text{drop } k)$; even $\text{pure} \circ \text{head} = \partial (\text{pure} \circ \text{last})$ where $\text{pure } x = [x]$.

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

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

- ◊ ‘Sum’ up the elements of the list, associating to the right.
- ◊ This function just replaces cons “`:`” and `[]` with \oplus 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:

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

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 — $h(x \oplus y) = h\ y \oplus h\ x$ — then h swaps folds: $h \cdot foldr (\oplus) e = foldl (\ominus) e'$ where $e' = h\ e$ and $x \ominus y = x \oplus h\ y$.

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”.

```
data D = C0 τ1 τ2 ... τm | C1 ... | Cn ... deriving Show
```

There are n constructors C_i that make *different* values of type D ; e.g., $C_0\ x_1\ x_2\ \dots\ x_m$ is a D -value whenever each x_i is a τ_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 C_i .

In-fact, we could have written `data D α1 α2 ... αk = ...`, so that we speak of “ D values *parameterised* by types α_i ”. E.g., “lists whose elements are of type α ” is defined by `data List α = Nil | Cons α (List α)` 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 *mix in* such a feature.

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* α each implement a definition of the *class method* `show`. The type of `show` is written `Show α => α -> String`: *Given an argument of type α, look in the global listing of Show instances, find the one for α, and use that*; if α has no `Show` instance, then we have a type error. One says “the type variable α has is *restricted* to be a `Show` instance” —as indicated on the left side of the “ \Rightarrow ” 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
<code>Show</code>	Show elements as strings, <code>show</code>
<code>Read</code>	How to read element values from strings, <code>read</code>
<code>Eq</code>	Compare elements for equality, <code>==</code>
<code>Num</code>	Use literals 0, 20, ..., and arithmetic +, *, -
<code>Ord</code>	Use comparison relations >, <, >=, <=
<code>Enum</code>	Types that can be listed, <code>[start .. end]</code>
<code>Monoid</code>	Types that model ‘(untyped) composition’
<code>Functor</code>	Type formers that model effectful computation
<code>Applicative</code>	Type formers that can sequence effects
<code>Monad</code>	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 `Num` class in Haskell could be defined as follows.

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

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 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!

-
- ◊ `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 give us: `fmap (f1 . f2 fn) = fmap f1 fmap fn` for `n ≥ 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`.

Applicative —Protecting against invalid input

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” -}
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  {-# MINIMAL pure, ((<*>) | liftA2) #-}

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

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

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 \approx 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 α` as an “effectful computation on α ”, then the above laws ensure `pure` creates an “effect free” context. E.g., if `f α = [α]` is considered “nondeterministic α -values”, then `pure` just treats usual α -values as nondeterministic but with no ambiguity, and `fs <*> xs` reads “if we nondeterministically have a choice `f` from `fs`, and we nondeterministically an `x` from `xs`, then we nondeterministically 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 `fmap` 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.)

Any expression built from the applicative methods can be transformed to the canonical form of “a pure function applied to effectful arguments”: `pure f <*> x1 <*> ... <*> xn` —The laws, as left-to-right rewrite rules, are the algorithm. Notice that the canonical form generalises `fmap` to `n`-arguments: Given `f :: α1 -> ... -> αn -> β` and `xi :: f αi`, we obtain an `(f β)`-value. The case of `n = 2` is called `liftA2`, and `n = 1` is just `fmap`.

Notice that `liftA2` 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
{- Associativity -} assoc1 <$> (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)`.

Do Notation

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 { $x_1 \leftarrow p_1$; ...; $x_n \leftarrow p_n$; pure e }` \approx `pure ($\lambda x_1 \dots x_n \rightarrow e$) <*> p_1 <*> ... <*> p_n` **provided** p_i does not mention x_j for $j < i$; but e may refer to all x_i . If any p_i mentions an earlier x_j , then we could not translate the `do`-notation into an applicative expression.

If `do { $x \leftarrow p$; $y \leftarrow qx$; pure e }` has `qx` being an expression **depending** on `x`, then we could say this is an abbreviation for `($\lambda x \rightarrow (\lambda y \rightarrow e) <$> qx) <$> p$` but this is of type `f (f β)`. Hence, to allow later computations to depend on earlier computations, we need a method `join :: f (f α) \rightarrow α` with which we define `do { $x \leftarrow p$; $y \leftarrow qx$; pure e }` \approx `join $ ~($\lambda x \rightarrow (\lambda y \rightarrow e) <$> 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 { $x \leftarrow p$; $y \leftarrow q$; 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.

Todo – Monad —“the programmable semicolon”

Coming soon ... See end of week of April 3rd, 2020 ...

Reads

- ◊ [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.*