Misc

Purity

- **A function is pure iff the same input always produces the same output and it has no side effects (no effects other than what it returns)**
  - Equivalent to **referential transparency**
  - Functions with return type `IO a` return an action of type `IO a` when evaluated — only cause effects when the returned action is executed in order to obtain a value of type `a`
- IO is pure — as it takes as input the entire real world and returns as output the entire real world it must be as it never takes the same input and everything that has occurred it returned

Writing complete haskell programs

- Each haskell file is called a module
- **The module named Whatever.hs must contain a line `module Whatever where`** — like Java filename needs to match the name in the file
- **Imports come after module … and before all the definitions**

Best Practices

- Separation of concerns
- Principle of least privilege
  - Don’t use IO if you don’t need to
  - When you do need to use IO, factor as much of the logic as possible out into pure functions
- Principle of least surprise
- Single responsibility principle
- Use libraries but don’t import a big library for a function you could’ve implemented yourself fairly easily
- Use explicit export lists to limit the interface provided to other code to ensures your code is used as intended
  - **Smart constructors — don’t export the constructor defined alongside the type, only export (the type itself and) a function that calls the actual constructor and checks arguments against predicates first**
- Use explicit import lists to import only the functions you need from modules you import
- Don’t write or use partial functions if at all possible — they’re the null pointers of Haskell, when they break everything they touch breaks and it's only a matter of time before they break
  - Use uncons instead of head or tail
  - Use !? instead of ! on maps
  - A function with a partial pattern match is a partial function

Polymorphism

- **Parametric polymorphism = when a function is able to adopt a range of types due to the use of unconstrained type variables in its type signature**
- **Ad-hoc polymorphism = when a function is able to adopt any one of a range types because it has been given separate definition for different types — type classes in type constraints of functions**
- Subtype polymorphism — type classes in type constraints on type class declarations

The basics

Evaluation

- Referential transparency is very useful
- Due to laziness, outermost expressions are typically evaluated first
- Function application has the highest precedence (10/9) and is left associative
- The $ operator carries out function application but has the lowest precedence and is right associative
- a ~ b ~ c ~ d
  -- if ~ is left associative, it is evaluated left to right
  ((a ~ b) ~ c) ~ d
  -- if ~ is right associative, it is evaluated right to left (sort of)
  a ~ (b ~ (c ~ d))
- A fixity declaration defines the associativity and precedence of an operator
  - infixr means right associative
  - infixl means left associative
  - infix means non-associative
  - Each of these followed by a number denoting the precedence (0-9) which is followed by the operator

Typing functions

- **Function application associates to the left** — f x y z === ((f x) y) z
- **Type arrows associate to the right** — f :: Int -> Int -> Int —> Int === f :: Int -> (Int -> (Int -> Int))
  - Informally this is a function that takes a three ints and returns an Int
  - Formally, this is a function that takes an Int returns a function that takes an Int that returns a function that takes an Int that returns an Int
- **Currying = the process of converting a function that takes multiple arguments into a series of functions that each take one argument**
- Functions are curried automatically unless they take their multiple arguments as a single argument of a tuple — partial function application comes for free
  - If `f :: a -> b -> c`, then `(f x) :: a -> c` and `(x f) :: b -> c`

Lists

- [] constructs an empty list
- : is the cons operator — **`h : t` prepends h to t**
- **All lists are made of a chain of : <u>ending</u> with []**

```
-- Equivalent to  "x = [3, 2, 1]"
-- Brackets are optional as cons is right associative
x = 3 : (2 : (1 : []))
```

- **Lists in Haskell** are singly linked lists (so chose operations carefully) and can **only contain data of one type**
- **Bindings in list comprehension generators are parsed left-to-right —** `[(x,y) | x <- [1,2,3], y <- [0..x] ] === [ (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3)]`
  - **Guards allow a predicate to be applied to the generator —** `evens = [x | x <- [0..], even x]`
  - Prefer composing higher order functions to writing complex list comprehensions

Higher order functions

- **Explicit recursion = A function calling itself**
- **Implicit recursion = A function passing itself to a function that will call it** (e.g. map, filter, foldl, foldr)
- **Higher-order function = a function that takes a function as an argument or returns a function**

Types

Types

- **A type built from a constructor with no paraters has cardinality 1**
- **A sum type is formed by separate types with | — takes the value of exactly one of the types**
  - Cardinality of a|b|... = cardinality of a + cardinality of b + …
- **A product type has a constructor with a non-zero number of parameters — takes the value of a combination of all the types**
  - Cardinality of a b c … = cardinality of a * cardinality of b * …
- **Tpes which have exactly one constructor and whose constructor takes exactly one parameter are called wrapper types** as it is really the type of the parameter
  - **Using `newtype` instead of `data` tells the compiler that a type is a wrapper type and hence certain optimizations can be made**
- **`type aName =` … defines an alias for a type e.g. `type String = [Char]` is in prelude**
- **Data structures are created by defining recursive data types**

Records

- `data Student = Student {firstName :: String, lastName :: String, enrolledModules :: [Module]}` is synaptic sugar for `data Student = Student String String [Module]` and some associated functions
  - A record brings into scope an accessor function for each field which takes in a record of that type and returns the value of the field — don't have multiple record types with shared field names in the same namespace!
- s {f1=v1, …} creates a new instance with the same record type as s with the fields f1… given values v1… and all other fields the same as s

Kinds

- At the value level we have constructors, functions that take values or functions and return a new value or function
- At the type level we have type constructors, functions that take types or type constructors and return a type or a type constructor
- Kinds are to types and type constructors as types are to values and constructors
- Kind signatures use :: and —> like other functions — *** denotes the kind of a type which takes zero type parameters** (a star cannot be filed in by a type with kind with arrows in, it can only be filled in by a concrete type)

```
String :: *
Maybe :: * -> *
Either :: * -> * -> *
```

  - **Number of arrows = number of type parameters (each of which must be a concrete type i.e. of kind *)**
- All fully applied data types, and all primitive types, are of kind *
- All arguments to functions are of kind *
- In `class Foldable t where foldr :: (a -> b -> b) -> b -> t a -> b` `t` must be of kind `* —> *` and a and b must be of kind `*`

Typeclasses

Semigroups and monoids

- **In maths, a semigroup is a set S equipped with an associative function from S^2 to S**
- **Haskell has a semigroup type class**

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

- **The semigroup operation for sets could be union or intersection, it is union (because it allows a monoid to be constructed)**
- **A monoid is a semigroup equipped with an identity element**
- **Haskell has a monoid typeclass — the identity element is called mempty**
  - **Laws**
    - **For all x, x <> mempty === x**
    - **For all x, mempty <> x === x**
    - **Associativity from semigroup**
- \+ and 0 or * and 1 are both possible monoids for numeric types
  - Data.Monoid contains newtype wrappers sum and product to allow you to specify which should be used
- **`foldMap :: Monoid m => (a -> m) -> t a -> m` applies a function to every element in a foldable structure and then combines them** using a foldr with <> and mempty

(Applicative) Functors

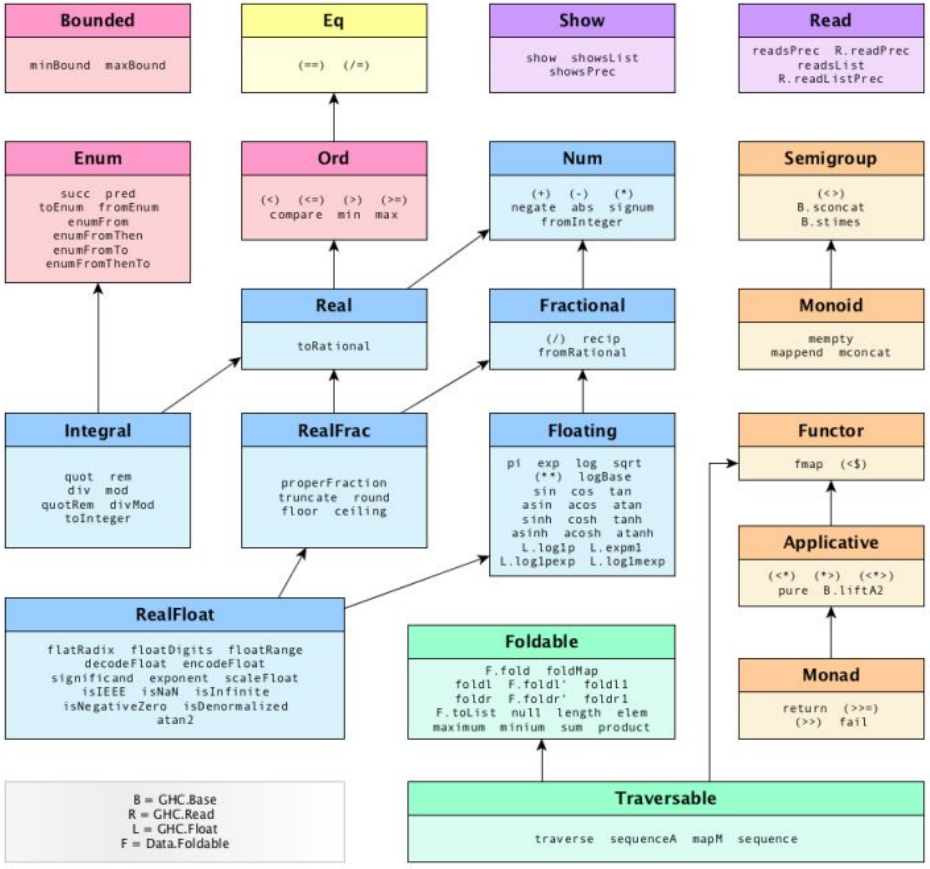
- **Members of functor are data containers that can be mapped over by functions**
  - <$> is the operator version of fmap
  - Functor applies a function to each element of a container/wrapped by a context without modifying the container/context
  - Functor is the wrapper not the function! (don't let mathematicians name things)
- **Members of applicative are functors that can be mapped over by functors that contain functions**
- Applicative is useful because a function that is partially applied to a functor will return a functor containing the new function
  - `(+) <$> (Just 2) <*> (Just 3) === Just(\y -> 2+y) <*> (Just 3) === Just (2+3) === Just 5`

Monads

- <$> passes a functor to a function that returns a value
- <*> passes a functor to a functor that contains a function that returns a value
- >>= passes a monadic value to a function that returns a monadic value
    - In some languages is implemented by applying fmap then unpacking both layers and repacking with one layer
- If we have some value that we wish to rise to the top in the event of an error we can use Either with the left being the type of the error value (e.g. a string containing an error message) and the right being the type of the value where the computation was successful

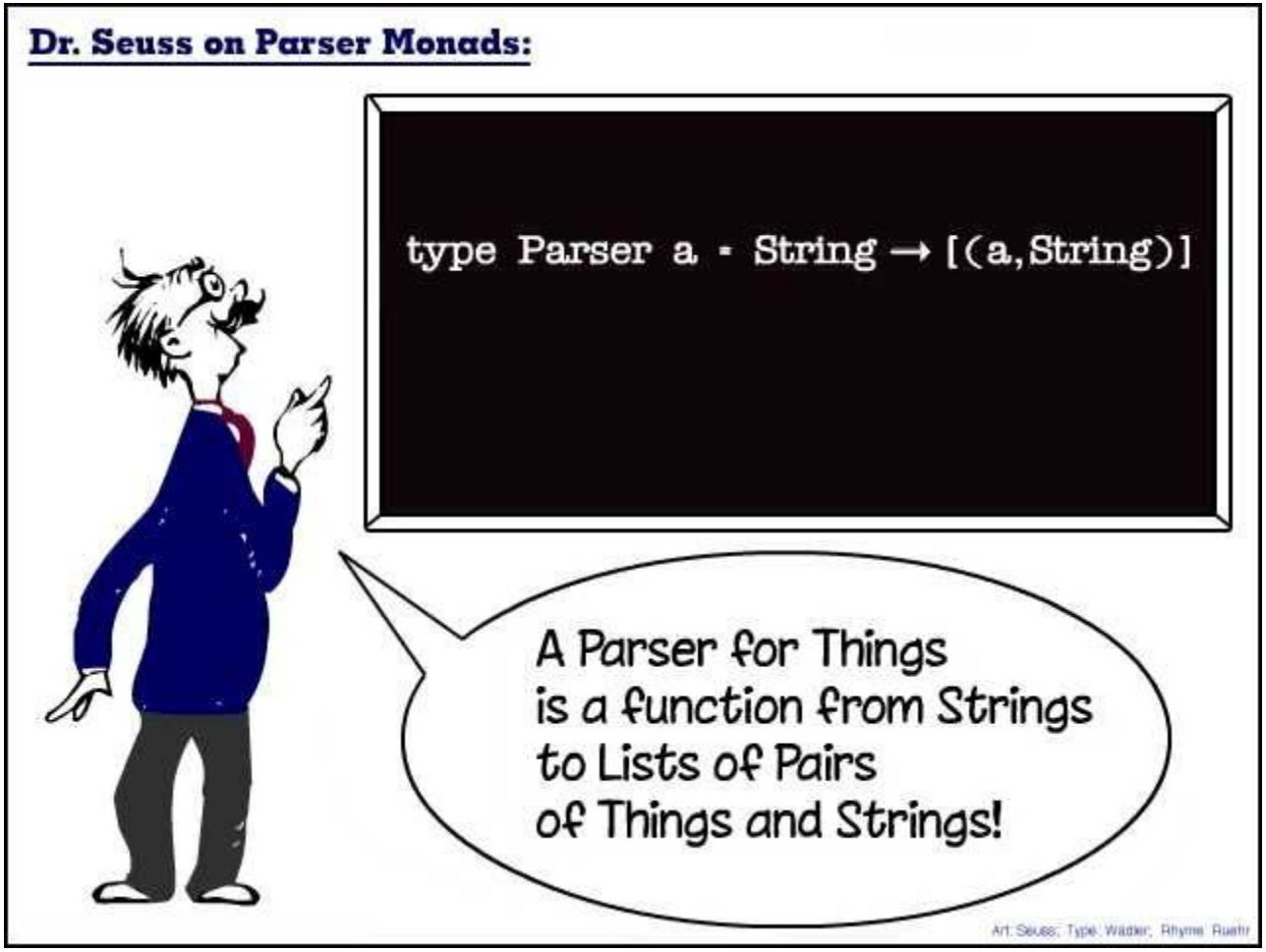
Typeclass hierarchy

- Type constraints can be placed in type class definitions as well as type class instance definitions — doing so is said to create a subtyping relationship



Parsing

Misc


Dr. Seuss on Parser Monads:
type Parser a = String → [(a,String)]
A Parser for Things
is a function from Strings
to Lists of Pairs
of Things and Strings!
Art: Seuss; Type: Wadler; Rhyme: Ruehr