Applicative Parsing

Julie Moronuki

January 10, 2017

Hour 1: Applicative

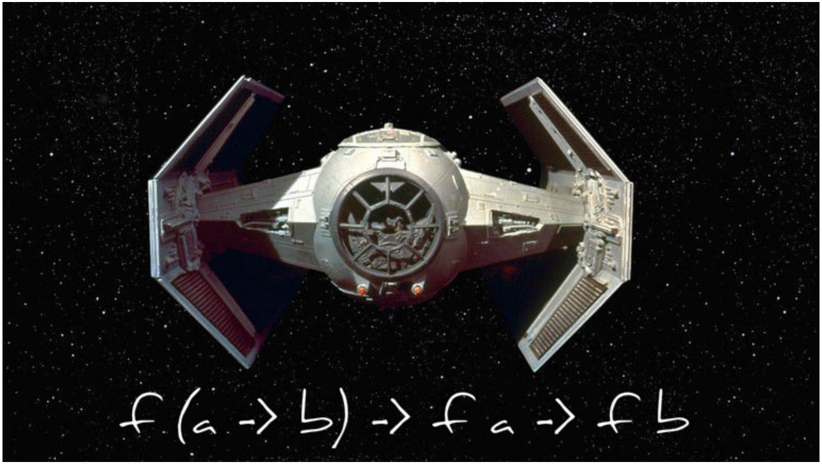

Figure 1: Tie Fighter of Doom

Functor

```haskell
fmap :: (a -> b) -> f a -> f b
```

Monad

- . . . is a kind of functor.
- but the `(a -> b)` of `fmap` has become an `(a -> f b)`

fmap vs bind

```
fmap :: Functor f => (a -> b) -> f a -> f b

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

flip bind

```
fmap :: Functor f => (a ->  b)  -> f a -> f b

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

Join

```haskell
join :: Monad m => m (m a) -> m a
```

Monad

Applicative

Figure 2: I have altered the Functor.

Applicative

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

Applicatives vs Monads

- context sensitivity
- composability (applicatives compose; monads need transformers)

Applicative vs Monad

```
doSomething = do
  a <- f
  b <- g
  c <- h
  pure (a, b, c)

doSomething' n = do
  a <- f n
  b <- g a
  c <- h b
  pure (a, b, c)
```

AccValidation

- like an Either, but accumulates error values
- cannot have a Monad instance

ApplicativeDo

- language extension
- allows use of `do` syntax with applicatives

Hour 2: Electric Boogaloo

In this hour, we'll be working on a small project with the optparse-applicative library.

Example

– stack new optex simple

Options.Applicative.Builder

Here are some basic argument types we can use: commands, flags, switches.

```
command :: String -> ParserInfo a -> Mod CommandFields a
```

Add a command to a subparser option.

Flag and flag'

```
flag :: a  -> a -> Mod FlagFields a -> Parser a
--      [1]   [2]           [3]                [4]
```

1. default value
2. active value
3. option modifier
4. Builder for a flag parser

Switch

```haskell
switch :: Mod FlagFields Bool -> Parser Bool
```

– flagEx.hs

helpful builders

```
subparser :: Mod CommandFields a -> Parser a

-- Builder for a command parser.

strArgument :: Mod ArgumentFields String -> Parser String

-- Builder for a String argument.

argument :: ReadM a -> Mod ArgumentFields a -> Parser a

-- Builder for an argument parser.
```

information we can provide about arguments

```
short :: HasName f => Char -> Mod f a

-- Specify a short name for an option.

long :: HasName f => String -> Mod f a

-- Specify a long name for an option.

metavar :: HasMetavar f => String -> Mod f a

-- Specify a metavariable for the argument.
```

Options.Applicative.Extra

```haskell
execParser :: ParserInfo a -> IO a
```

Run a program description.
Parse command line arguments. Display help text and exit if any parse error occurs.