

# Applicative Regular Expressions w/ the Free Alternative

Justin Le (@mstk)

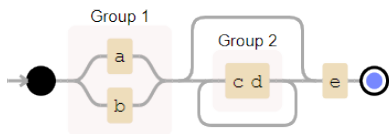
Compose Conference 2019, June 24

# Preface

Slide available at <https://talks.jle.im>.

# Regular Expressions

$(a|b)(cd)^*e$

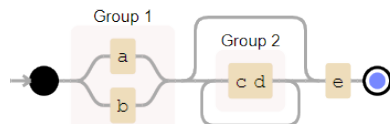


# Regular Expressions

$(a|b)(cd)^*e$

Matches:

- ▶ ae
- ▶ acdcdcde
- ▶ bcde

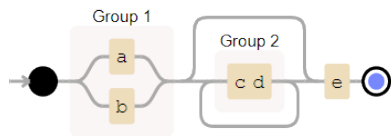


Doesn't match:

- ▶ acdcd
- ▶ abcde
- ▶ bce

# “Captures”

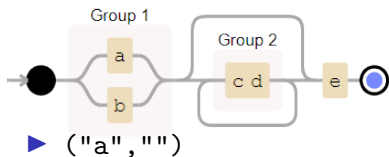
$(a|b)(cd)^*e$



# “Captures”

$(a|b)(cd)^*e$

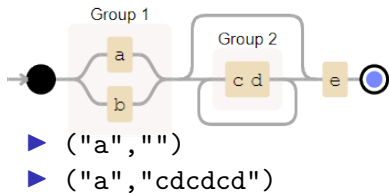
- ▶  $ae \rightarrow$
- ▶  $acdcdcde \rightarrow$
- ▶  $bcde \rightarrow$



# “Captures”

$(a|b)(cd)^*e$

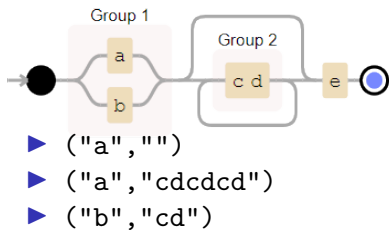
- ▶  $ae \rightarrow$
- ▶  $acdcdcde \rightarrow$
- ▶  $bcde \rightarrow$



# “Captures”

$(a|b)(cd)^*e$

- ▶  $ae \rightarrow$
- ▶  $acdcdcde \rightarrow$
- ▶  $bcde \rightarrow$





# Applicative Regular Expressions

# Applicative Regular Expressions

“Type-indexed” regular expressions.

```
type RegExp a  
    -- ^ type of "result"
```

# Applicative Regular Expressions

“Type-indexed” regular expressions.

```
type RegExp a
    -- ^ type of "result"

char    :: Char    -> RegExp Char
string  :: String  -> RegExp String
```

# Applicative Regular Expressions

“Type-indexed” regular expressions.

```
type RegExp a
    -- ^ type of "result"

char    :: Char    -> RegExp Char
string  :: String  -> RegExp String

runRegex :: RegExp a -> String -> Maybe a
```

## Applicative Regular Expressions

```
char    :: Char      -> RegExp Char
string  :: String    -> RegExp String
(<|>)   :: RegExp a -> RegExp a -> RegExp a
many    :: RegExp a -> RegExp [a]
```

```
myRegex :: RegExp (Char, [String])
myRegex = (,) <$> (char 'a' <|> char 'b')
           <*> many (string "cd")
           <*> char 'e'
```

## Applicative Regular Expressions

```
char    :: Char      -> RegExp Char
string  :: String    -> RegExp String
(<|>)   :: RegExp a -> RegExp a -> RegExp a
many    :: RegExp a -> RegExp [a]

myRegex :: RegExp (Char, [String])
myRegex = (,) <$> (char 'a' <|> char 'b')
           <*> many (string "cd")
           <*> char 'e'

runRegex myRegex :: String -> Maybe (Char, [String])
```

## Applicative Regular Expressions

```
runRegex myRegex "ae"
```

```
Just ('a', [])
```

```
runRegex myRegex "acdcdcde"
```

```
Just ('a', ["cd","cd","cd"])
```

```
runRegex myRegex "bcde"
```

```
Just ('b', ["cd"])
```

```
runRegex myRegex "acdcd"
```

```
Nothing
```

## Applicative Regular Expressions

```
myRegexp2 :: RegExp (Bool, Int)
myRegexp2 = (,) <$> ((False <$ char 'a') <|> (True <$ char
    <*> fmap length (many (string "cd")))
    <*> char 'e')
```



## Applicative Regular Expressions

```
myRegex2 :: RegExp (Bool, Int)
myRegex2 = (,) <$> ((False <$ char 'a') <|> (True <$ char
    <*> fmap length (many (string "cd")))
    <*> char 'e')
```

```
runRegex myRegex2 "ae"
Just (False, 0)
```

```
runRegex myRegex2 "acdcdcde"
Just (False, 3)
```

```
runRegex myRegex2 "bcde"
Just (True, 1)
```

What's so Regular about Regexp?

# What's so Regular about Regexp's?

## Regular Language Base Members

1. Empty language: Always fails to match
2. Empty string: Always succeeds, consumes nothing
3. Literal: Matches and consumes a given char

# What's so Regular about Regexp's?

## Regular Language Base Members

1. Empty language: Always fails to match
2. Empty string: Always succeeds, consumes nothing
3. Literal: Matches and consumes a given char

## Regular Language Operations

1. Concatenation:  $RS$ , sequence one after the other
2. Alternation:  $R|S$ , one or the other
3. Kleene Star:  $R^*$ , the repetition of  $R$

# An Alternative Perspective

## An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b
```

## An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b
```

**Components of a Regular Language**

1. Empty language: empty

```
class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```



# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b
```

## Components of a Regular Language

1. Empty language: empty
2. Empty string: pure x

```
class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b
```

## Components of a Regular Language

1. Empty language: empty
2. Empty string: pure x
3. Literal: ???

```
class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

## Components of a Regular Language

1. Empty language: empty
2. Empty string: pure x
3. Literal: ???
4. Concatenation: <\*>

# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

## Components of a Regular Language

1. Empty language: empty
2. Empty string: pure x
3. Literal: ???
4. Concatenation: <\*>
5. Alternation: <|>

# An Alternative Perspective

```
class Functor f => Applicative f where
  -- | Always succeed, consuming nothing
  pure  :: a -> f a
  -- | Concatenation
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Alternative f where
  -- | Always fails to match
  empty :: f a
  -- | Alternation
  (<|>) :: f a -> f a -> f a
  -- | Reptition
  many  :: f a -> f [a]
```

## Components of a Regular Language

1. Empty language: empty
2. Empty string: pure x
3. Literal: ???
4. Concatenation: <\*>
5. Alternation: <|>
6. Repetition: many

# Functor combinator-style

- ▶ Define a primitive type

```
type Prim a
```

# Functor combinator-style

- ▶ Define a primitive type

```
type Prim a
```

- ▶ Add the structure you need

# Functor combinator-style

- ▶ Define a primitive type

```
type Prim a
```

- ▶ Add the structure you need

- ▶ If this structure is from a typeclass, use the free structure of that typeclass



## Easy as 1, 2, 3

```
-- | Free Alternative, from 'free'
data Alt :: (Type -> Type) -> (Type -> Type)
      -- ^ take a Functor; ^ return a Functor
instance Functor      (Alt f)
instance Applicative (Alt f)
instance Alternative  (Alt f)

liftAlt :: Prim a -> Alt Prim
```

## Easy as 1, 2, 3

```
-- | Free Alternative, from 'free'
data Alt :: (Type -> Type) -> (Type -> Type)
    -- ^ take a Functor; ^ return a Functor
instance Functor      (Alt f)
instance Applicative (Alt f)
instance Alternative (Alt f)

liftAlt :: Prim a -> Alt Prim

data Prim a = Prim Char a
    deriving Functor

type RegExp = Alt Prim

char :: Char -> RegExp Char
char c = liftAlt (Prim c c)
```

# Unlimited Power

```
empty :: RegExp a
pure  :: a -> RegExp a
char  :: Char -> RegExp Char
(<*>) :: RegExp (a -> b) -> RegExp a -> RegExp b
(<|>) :: RegExp a -> RegExp a -> RegExp a
many  :: RegExp a -> RegExp [a]
```

# Unlimited Power

```
empty :: RegExp a
pure  :: a -> RegExp a
char  :: Char -> RegExp Char
(<*>) :: RegExp (a -> b) -> RegExp a -> RegExp b
(<|>) :: RegExp a -> RegExp a -> RegExp a
many  :: RegExp a -> RegExp [a]

string :: String -> RegExp String
string = traverse char

digit :: RegExp Int
digit = asum [ intToDigit i <$ char i | i <- [0..9] ]
```

# Parsing

## Options

1. *Interpret* into an `Alternative` instance, “offloading” the logic

# Parsing

## Options

1. *Interpret* into an `Alternative` instance, “offloading” the logic
  - ▶ Analogy: Process a list using `foldMap`

# Parsing

## Options

1. *Interpret* into an `Alternative` instance, “offloading” the logic
  - ▶ Analogy: Process a list using `foldMap`
2. Direct pattern match on structure constructors (Haskell 101)

# Parsing

## Options

1. *Interpret* into an `Alternative` instance, “offloading” the logic
  - ▶ Analogy: Process a list using `foldMap`
2. Direct pattern match on structure constructors (Haskell 101)
  - ▶ Analogy: Process a list using pattern matching and recursion



What is freeness?

## What is freeness?

```
class Semigroup m where
  (<>)    :: m -> m -> m
class Monoid m where
  mempty  :: m

type FreeMonoid = [] :: Type          -> Type
           -- ^ take a type; ^ return a type
instance Semigroup (FreeMonoid a)
instance Monoid    (FreeMonoid a)

injectFM :: a -> FreeMonoid a
runFM    :: Monoid m => (a -> m) -> (FreeMonoid a -> m)

runFM f
  -- ^ substitute injectFM for f
```

## What is freeness?

```
class Semigroup m where
  (<>)    :: m -> m -> m
class Monoid m where
  mempty  :: m

type FreeMonoid = [] :: Type          -> Type
           -- ^ take a type; ^ return a type
instance Semigroup (FreeMonoid a)
instance Monoid    (FreeMonoid a)

injectFM :: a -> FreeMonoid a
runFM    :: Monoid m => (a -> m) -> (FreeMonoid a -> m)

runFM f
  -- ^ substitute injectFM for f

(:[])   :: a -> [a]
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
```

## What is freeness?

```
myMon :: FreeMonoid Int
```

```
myMon = injectFM 1 <> mempty <> injectF1 2 <> injectF1 3 <>
```

## What is freeness?

```
myMon :: FreeMonoid Int
myMon = injectFM 1 <> mempty <> injectF1 2 <> injectF1 3 <> ...

foldMap Sum myMon      -- mempty = 0, <> = +
Sum 1 + Sum 0 + Sum 2 + Sum 3 + Sum 4
Sum 10
```

## What is freeness?

```
myMon :: FreeMonoid Int
myMon = injectFM 1 <> mempty <> injectF1 2 <> injectF1 3 <> injectF1 4

foldMap Sum myMon      -- mempty = 0, <> = +
Sum 1 + Sum 0 + Sum 2 + Sum 3 + Sum 4
Sum 10

foldMap Product myMon  -- mempty = 1, <> = *
Product 1 * Product 1 * Product 2 * Product 3 * Product 4
Product 24
```

## What is freeness?

```
myMon :: FreeMonoid Int
myMon = injectFM 1 <> mempty <> injectF1 2 <> injectF1 3 <> injectF1 4

foldMap Sum myMon      -- mempty = 0, <> = +
Sum 1 + Sum 0 + Sum 2 + Sum 3 + Sum 4
Sum 10

foldMap Product myMon  -- mempty = 1, <> = *
Product 1 * Product 1 * Product 2 * Product 3 * Product 4
Product 24

foldMap Max myMon      -- mempty = minBound, <> = max
Max 1 `max` Max minBound `max` Max 2 `max` Max 3 `max` Max 4
Max 4
```

## What is freeness?

```
type Alt f a

-- / Analog to (:[])
liftAlt :: f a -> Alt f a

-- / Analog to foldMap
runAlt  :: Alternative g
        => (forall b. f a -> g a)
        -> (Alt f a -> g a)

runAlt f
    -- ^ substitute liftAlt for f
```



# Goal

Find an Alternative where:

- ▶ Prim a = consumption

# Goal

Find an Alternative where:

- ▶ Prim a = consumption
- ▶  $\langle * \rangle$  = sequencing consumption

# Goal

Find an Alternative where:

- ▶ Prim a = consumption
- ▶  $\langle * \rangle$  = sequencing consumption
- ▶  $\langle | \rangle$  = backtracking

# Hijacking StateT

StateT [Char] Maybe

- ▶ Prim a can be interpreted as *consumption* of state
- ▶ <\*> sequences consumption of state
- ▶ <|> is backtracking of state

## Hijacking StateT

```
processPrim :: Prim a -> StateT String Maybe a
processPrim (Prim c x) = do
  d:ds <- get           -- ^ match on stream
  guard (c == d)        -- ^ fail unless match
  put ds                 -- ^ update stream
  return x              -- ^ return result value

matchPrefix :: Regexp a -> String -> Maybe a
matchPrefix re = evalStateT (runAlt processPrim re)
```

This works?

## This works?

Yes!

```
matchPrefix myRegex2 "ae"
```

```
Just (False, 0)
```

```
matchPrefix myRegex2 "acdcdcde"
```

```
Just (False, 3)
```

```
matchPrefix myRegex2 "bcde"
```

```
Just (True, 1)
```

What just happened?



## What just happened?

```
data Prim a = Prim Char a
  deriving Functor
```

```
type RegExp = Alt Prim
```

```
matchPrefix :: RegExp a -> String -> Maybe a
matchPrefix re = evalStateT (runAlt processPrim re)
  where
    processPrim (Prim c x) = do
      d:ds <- get
      guard (c == d)
      put ds
      pure x
```

Fits in a tweet!

## What just happened?

1. Offload Alternative functionality to StateT: empty, <\*>, pure, empty, many.

## What just happened?

1. Offload Alternative functionality to StateT: `empty`, `<*>`, `pure`, `empty`, `many`.
2. Provide Prim-processing functionality with `processPrim`: `liftAlt`.

# What do we gain?

1. Interpretation-invariant structure

# What do we gain?

1. Interpretation-invariant structure
2. Actually meaningful types

# What do we gain?

1. Interpretation-invariant structure

2. Actually meaningful types

▶ `StateT String Maybe` is **not** a regular expression type.

```
notARegexp :: StateT String Maybe ()
```

```
notARegexp = put "hello"           -- no regular expression
```

# What do we gain?

1. Interpretation-invariant structure

2. Actually meaningful types

▶ `StateT String Maybe` is **not** a regular expression type.

```
notARegexp :: StateT String Maybe ()
```

```
notARegexp = put "hello"           -- no regular expression
```

▶ `Alt Prim` **is** a regular expression type.

## Direct matching

```
newtype Alt f a = Alt { alternatives :: [AltF f a] }
```

```
data AltF f a = forall r. Ap (f r) (Alt f (r -> a))  
              | Pure a
```



## Direct matching

```
newtype Alt f a = Alt { alternatives :: [AltF f a] }
```

```
data AltF f a = forall r. Ap (f r) (Alt f (r -> a))  
              |              Pure a
```

```
-- | Chain of </>s
```

```
newtype Alt f a  
  = Choice (AltF f a) (Alt f a) -- ^ c  
  | Empty -- ^ n
```

```
-- | Chain of <*>s
```

```
data AltF f a  
  = forall r. Ap (f r) (Alt f (r -> a)) -- ^ c  
  | Pure a -- ^ n
```

## Direct Matching

```
matchAlts :: RegExp a -> String -> Maybe a
matchAlts (Alt res) xs = asum [ matchChain re xs | re <- res ]
```

## Direct Matching

```
matchAlts :: RegExp a -> String -> Maybe a
matchAlts (Alt res) xs = asum [ matchChain re xs | re <- res ]

matchChain :: AltF Prim a -> String -> Maybe a
matchChain (Ap (Prim c x) next) cs = _
matchChain (Pure x)                cs = _
```

## One game of Tetris later...

```
matchChain :: AltF Prim a -> String -> Maybe a
matchChain (Ap (Prim c x) next) cs = case cs of
  [] -> Nothing
  d:ds | c == d -> matchAlts ((\$ x) <\$> next) ds
        | otherwise -> Nothing
matchChain (Pure x) _ = Just x
```

This works?

## This works?

Yes!

```
matchChain myRegex2 "ae"
```

```
Just (False, 0)
```

```
matchChain myRegex2 "acdcdcde"
```

```
Just (False, 3)
```

```
matchChain myRegex2 "bcde"
```

```
Just (True, 1)
```

## What do we gain?

- ▶ First-class program rewriting, Haskell 101-style

## What do we gain?

- ▶ First-class program rewriting, Haskell 101-style
- ▶ **Normalizing** representation



## What do we gain?

- ▶ First-class program rewriting, Haskell 101-style
- ▶ **Normalizing** representation
  - ▶ Equivalence in meaning = equivalence in structure

## What do we gain?

- ▶ First-class program rewriting, Haskell 101-style
- ▶ **Normalizing** representation
  - ▶ Equivalence in meaning = equivalence in structure

## What do we gain?

- ▶ First-class program rewriting, Haskell 101-style
- ▶ **Normalizing** representation
  - ▶ Equivalence in meaning = equivalence in structure

```
-- | Not regularizing
```

```
data RegExp a = Empty
              | Pure a
              | Prim Char a
              | forall r. Seq (RegExp a) (RegExp (r -> a))
              | Union (RegExp a) (RegExp a)
              | Many (RegExp a)
```

```
-- a/(b/c) /= (a/b)/c
```

# Free your mind

Is this you?

*“My problem is modeled by some (commonly occurring) structure over some primitive base.”*

- ▶ Use a “functor combinator”!

# Free your mind

Is this you?

*“My problem is modeled by some (commonly occurring) structure over some primitive base.”*

- ▶ Use a “functor combinator”!
- ▶ If your structure comes from a typeclass, use a free structure!

## Further Reading

- ▶ Blog post:  
<https://blog.jle.im/entry/free-applicative-regexp.html>
- ▶ Functor Combinatorpedia:  
<https://blog.jle.im/entry/functor-combinatorpedia.html>
- ▶ *free*: <https://hackage.haskell.org/package/free>
- ▶ *functor-combinators*:  
<https://hackage.haskell.org/package/functor-combinators>
- ▶ Slides: <https://talks.jle.im/composeconf-2019/>