Language Engineering | Lecture 15.

The functions some and many are defined in terms of each other. ~~This is the same principal as~~ This is achieved by mutual recursion, which is the same principle behind odd and even:

```
even :: Int → Bool
even 0 = True
even n = not (odd (n-1))

odd :: Int → Bool
odd 0 = False
odd n = not (even (n-1))
```

We could understand this by simply following the definitions. We can also inline:

```
even 0 = True
even n = not ~~(odd~~ (λx →
    case x of
        0 → False
        m → not (even m-1))
```

We want to understand <$> and <*>, since they are used often in parser combinators.

Remember map and fmap:

```
map :: (a -> b) -> [a] -> [b]
fmap :: (a -> b) -> f a -> f b
```

Let's compare this with (<$>)

```
(<$>) :: (a -> b) -> f a -> f b
```

This allows us to write

```
f <$> xs
```

instead of

```
fmap f xs
```

Now compare (<$>) with (<*>):
(this time, I won't hide the class constraints)

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

We had several instances of (<*>) defined in the last lecture, we're now ready for the (<*>) for Parser.

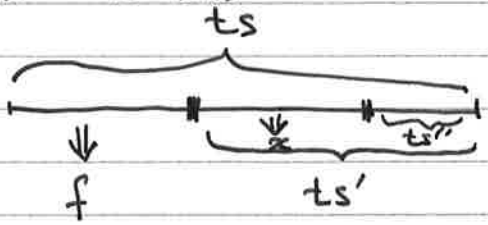
```
instance Functor Parser where
  fmap f (Parser p) = Parser (λts -> [(ts', f x)
                                     | (ts', x) <- p ts])
```

p : String -> [(String, a)]

This creates a new parser where the parsed values are transformed by the function f.

```
instance Applicative Parser where
  -- pure :: a -> Parser a
     pure x = produce x
  -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
     Parser pf <*> Parser px =
       Parser (λts -> [(ts'', f x) | (ts', f) <- pf ts
                                   , (ts'', x) <- px ts'
                                   ])
```

The result of Parser pf <*> Parser px is a bit like this:



f x will be the value of type b
ts'' is what is left over

Let's look at the type of some more closely:

```
some :: Parser a → Parser [a]
some v = (:) <$> v <*> many v
```