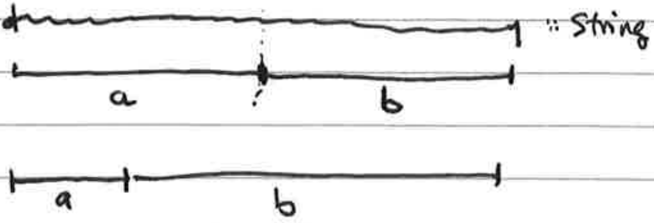
Language Engineering | Lecture 14

newtype Parser a :: Parser (String → [(String, a)])



(>>=) :: Parser a → (a → Parser b) → Parser b

px >>= f is a parser that first uses px to parse a value x :: a. The value x is used as a parameter to the function f, such that f x :: Parser b. We apply f x to the remaining tokens that were not consumed when making x.
NB. There may have been more than one result (x, ts) from applying the parser px, so we may have different f x combinations and final outputs.

```
oneOf :: [Char] → Parser Char
oneOf xs = satisfy (λx→ x'elem'xs)
```

```
alpha :: Parser Char
alpha = satisfy isAlpha
```

isAlpha :: Char → Bool
from the prelude

or

```
alpha :: Parser Char
alpha = oneOf(['a'..'z'] ++ ['A'..'Z'])
```

from prelude

```
nat :: Parser Integer
nat = read <$> some (satisfy isDigit)
```

:: String
→ Integer

mysteries
so far.

we understand

The parser some :: Parser a → Parser [a] is equivalent to the + operation in BNF, where some p is the same as p+. It is related to the parser many :: Parser a → Parser [a], which is where many p corresponds to p* in BNF.

some :: Alternative f ⇒ f a → f [a]
which specializes to:
some :: Parser a → Parser [a]

$$\text{Some } v = \underbrace{(:)}_{a \to [a] \to [a]} \; \text{<\$>} \; \underbrace{v}_{\text{Parser } a} \; \text{<*>} \; \underbrace{\text{many } v}_{\text{Parser } [a]}$$

many :: Alternative ⇒ fa → f [a]

$$\text{many } v = \underbrace{\text{some } v}_{\text{Parser } [a]} \; \text{<|>} \; \underbrace{\text{produce } [\,]}_{\text{Parser } [a]}$$

To understand some and many, we need to understand: <*>, <|>, and <$>.

The simplest is: (<|>).
p <|> q . This is an example of Alternative:

```
class Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

This has to satisfy some laws:

```
empty <|> q = q
p  <|> empty = p
```

We won't force associativity i.e. ((p <|> q) <|> r
= p <|> (q <|> r))

Instances of this exist in Maybe and []:

```
instance Altenative Maybe where
  empty = Nothing

  Just x <|> q = Just x
  Nothing <|> q = q
```

```
instance Alternative [] where
   empty  =  []
   xs <|> ys  =  xs ++ ys
```

The parser instance is a bit more complex:

```
  instance Alternative Parser where
     empty  =  failure
     (<|>)  =  orElse
```

```
 orElse :: Parser a → Parser a → Parser a
orElse (Parser px) (Parser py) = Parser (λts →
     case px ts of
        []  →  py ts
        xs  →  xs     )
        ↑
       :: [(String,a)]
```

The result of orElse px py is a parser that first tries to parce px, but if it fails, it tries py instead. If it succeeds then it is px.

The <*> is a member of the Applicative class:

```
class Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b.
```

pronounced 'ap' for apply.

```
instance Applicative [] where
    -- pure :: a->[a]
    pure x = [x]
    fs <*> xs = [ f x | f <- fs, x <- xs ]
```

--(<*>) :: [a->b] -> [a] -> [b]

this applies each function in fs to each value in xs.

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure x = Just x
    -- (<*>) :: Maybe (a->b) -> Maybe a -> Maybe b
    ~~Nothing <*> mx~~
    Just f <*> Just x = Just (f x)
    _      <*> _      = Nothing.
```