Lecture 4: More Parser Combinators

Talen en Compilers 2023-2024, period 2

Lawrence Chonavel

Department of Information and Computing Sciences, Utrecht University

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Recap: Parser Combinators

```
type Parser a = String -> [(a,String)]
```

Recap: Parser Combinators

```
type Parser a = String -> [(a,String)]

parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

Recap: Parser Combinators

```
type Parser a = String -> [(a,String)]

parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

<$> ::          (a -> b) -> Parser a -> Parser b
<*> :: Parser (a -> b) -> Parser a -> Parser b
```

Recap: Parser Combinators

```
type Parser a = String -> [(a,String)]

parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

<$> ::          (a -> b) -> Parser a -> Parser b
<*> :: Parser (a -> b) -> Parser a -> Parser b

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

Recap: Parser Combinators

```
type Parser a = String -> [(a,String)]

parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

<$> ::          (a -> b) -> Parser a -> Parser b
<*> :: Parser (a -> b) -> Parser a -> Parser b

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

👉 `parseDay :: Parser Int`
`parseMonth :: Parser Month`

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parseDay

```
parseDay :: Parser Int
```

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "👅" ❌
```

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "👅" ❌

parseDay "01" ✅
parseDay "00001" ✅
```

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "👅" ❌

parseDay "01" ✅
parseDay "00001" ✅

parseDay "The first of" ❓
parseDay "-1" ❓
parseDay "5000" ❓
```

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "😈" ❌

parseDay "01" ✅
parseDay "00001" ✅

parseDay "The first of" ❓
parseDay "-1" ❓
parseDay "5000" ❓

parseDay "31" ❓
```

parseDay

```
parseDay :: Parser Int

parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "👅" ❌

parseDay "01" ✅
parseDay "00001" ✅

parseDay "The first of" ❌
parseDay "-1" ❌
parseDay "5000" ✅

parseDay "31" ✅
```

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝
  - Error massage quality

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝
  - Error massage quality
- “29 Feb” ❓

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝
  - Error massage quality
- “29 Feb” ❓
- 😈 Dates are hard

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝
  - Error massage quality
- “29 Feb” ❓
- 😈 Dates are hard
  - codeblog.jonskeet.uk/2015/05/05/

Why not “31 Dec” ✅, “31 Feb” ❌ ?

- Better to parse then filter
  - Parser complexity 🍝
  - Validator complexity 🍝
  - Error massage quality
- “29 Feb” ❓
- 😈 Dates are hard
  - codeblog.jonskeet.uk/2015/05/05/
  - xkcd.com/1179/

parseDay

```
parseDay :: Parser Int
parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "😈" ❌

parseDay "01" ✅
parseDay "00001" ✅

parseDay "The first of" ❌
parseDay "-1" ❌

parseDay "5000" ✅
parseDay "31" ✅
```

parseDay

```
parseDay :: Parser Int
parseDay "15" ✅
parseDay "2" ✅

parseDay "haha nope" ❌
parseDay "👅" ❌

parseDay "01" ✅
parseDay "00001" ✅

parseDay "The first of" ❌
parseDay "-1" ❌

parseDay "5000" ✅
parseDay "31" ✅

parseDay = parsePositiveInt
```

parsePositiveInt

```
parsePositiveInt :: Parser Int

parsePositiveInt "1" ✅
parsePositiveInt "52" ✅
parsePositiveInt "999999999" ✅
parsePositiveInt "01" ✅
```

parsePositiveInt

```
parsePositiveInt :: Parser Int

parsePositiveInt "1" ✅
parsePositiveInt "52" ✅
parsePositiveInt "999999999" ✅
parsePositiveInt "01" ✅

parsePositiveInt "0" ❌
parsePositiveInt "-3" ❌
```

parsePositiveInt

```
parsePositiveInt :: Parser Int

parsePositiveInt "1" ✅
parsePositiveInt "52" ✅
parsePositiveInt "999999999" ✅
parsePositiveInt "01" ✅

parsePositiveInt "0" ❌
parsePositiveInt "-3" ❌

parsePositiveInt "1e10" ❌
parsePositiveInt "0xB33F" ❌
```

`parsePositiveInt` implementation

```
parsePositiveInt :: Parser Int

parsePositiveInt = ???
```

parsePositiveInt implementation

```
parsePositiveInt :: Parser Int

parsePositiveInt = ???

type Parser a = String -> [(a,String)] -- reminder
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt = ???

type Parser a = String -> [(a,String)] -- reminder
```

parsePositiveInt implementation

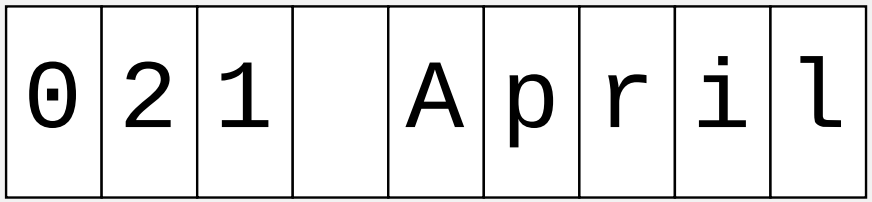
```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt = ???
```

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = ???
```

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = ???
```

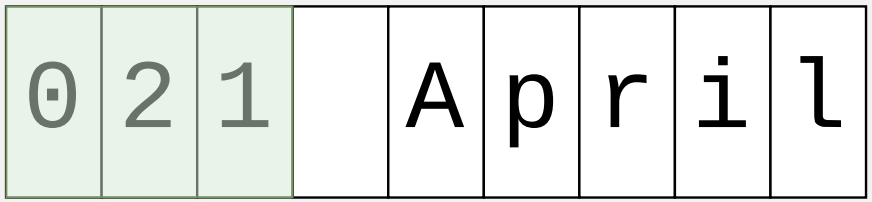


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = ???
```



Universiteit Utrecht

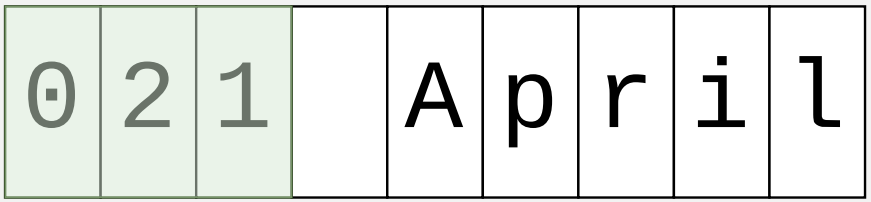
[Faculty of Science
Information and Computing
Sciences]

`parsePositiveInt` implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = ???
```

parsePositiveInt

("021 April")



parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = ???
```

parsePositiveInt

("021 April")

== [(21," April")]

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???,???)]
```

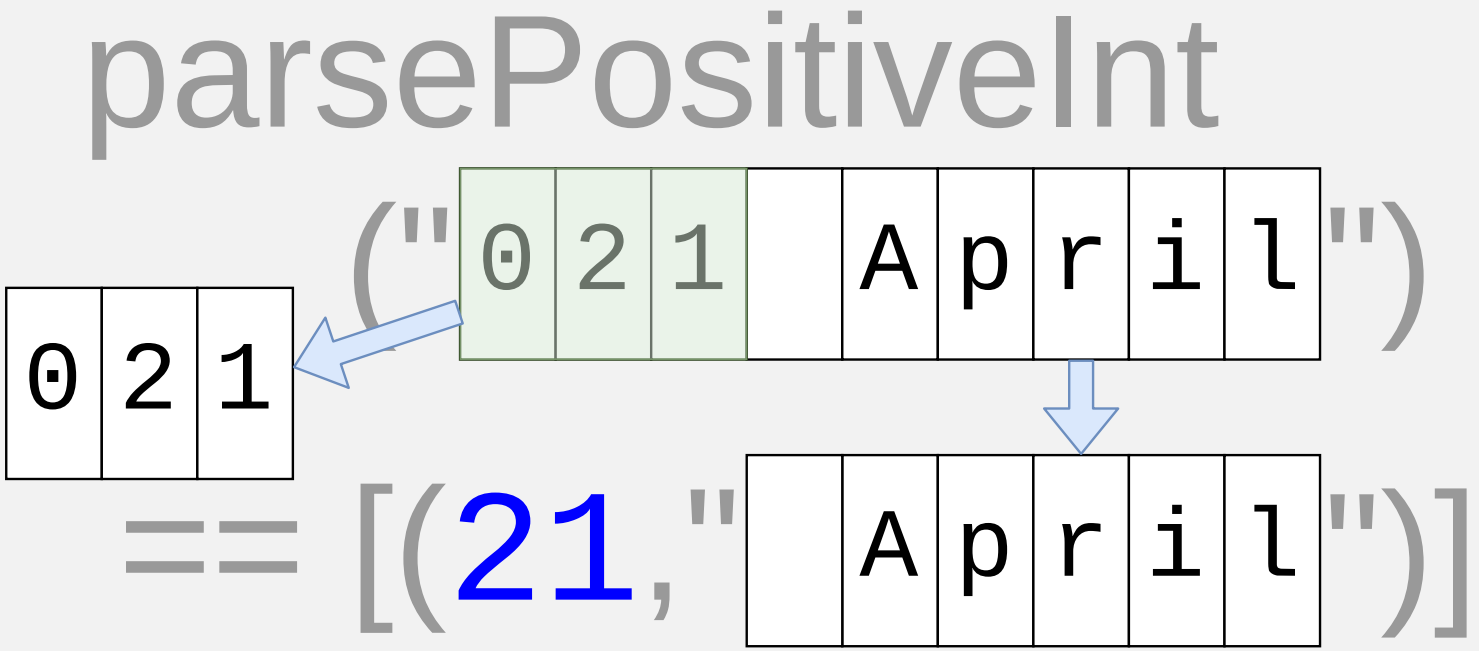
parsePositiveInt

("021 April")

== [(21," April")]

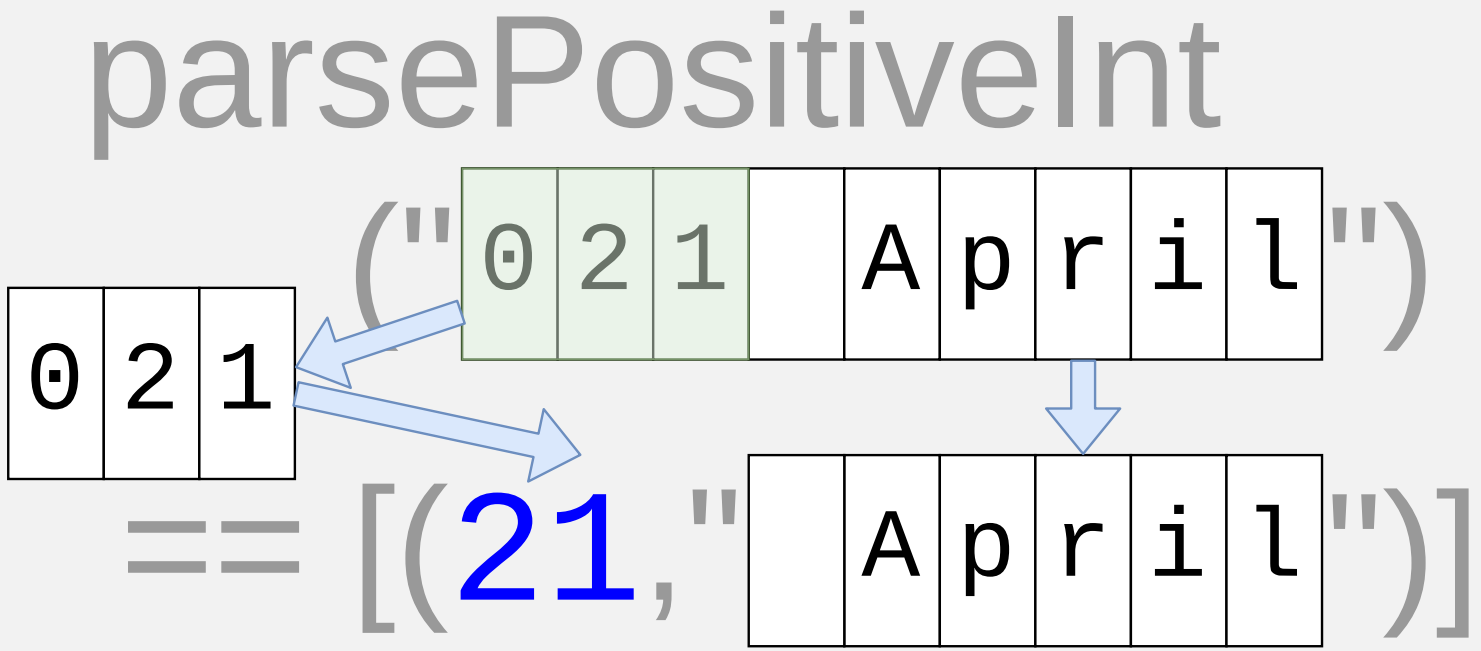
parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???,???)]
```



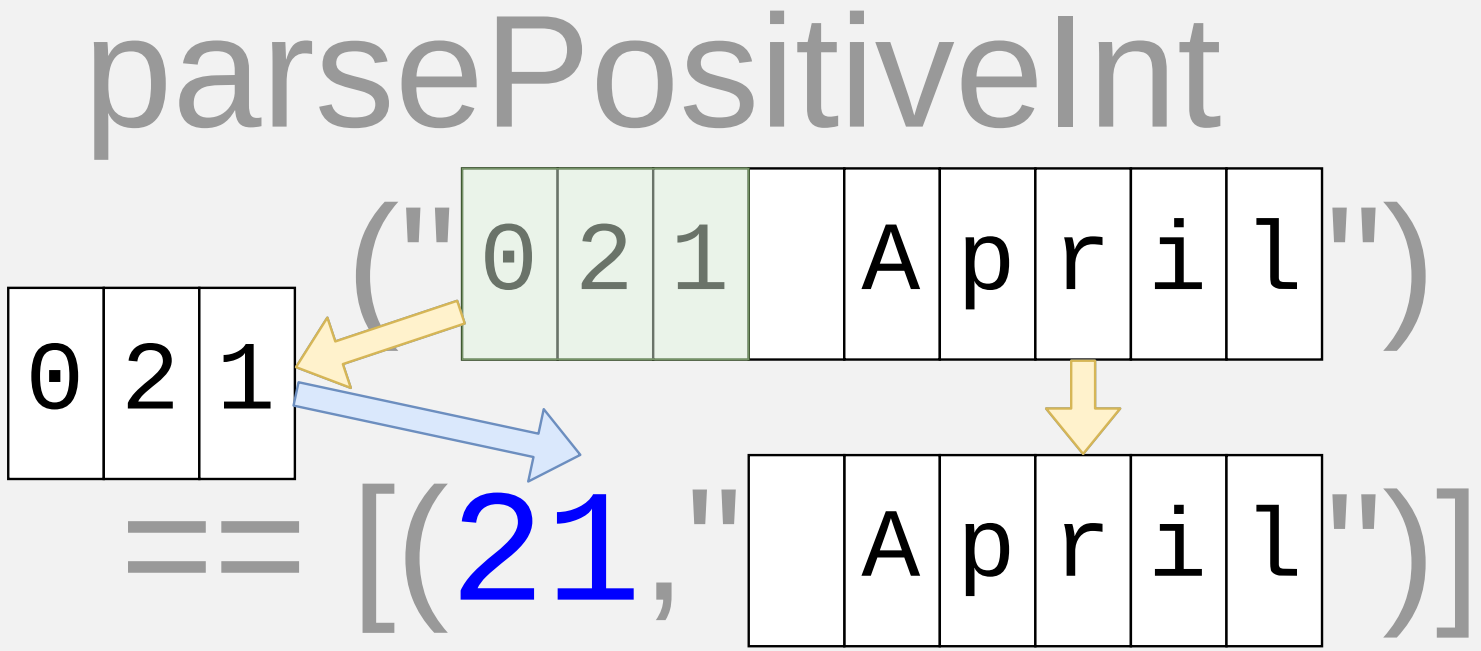
parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???,???)]
```



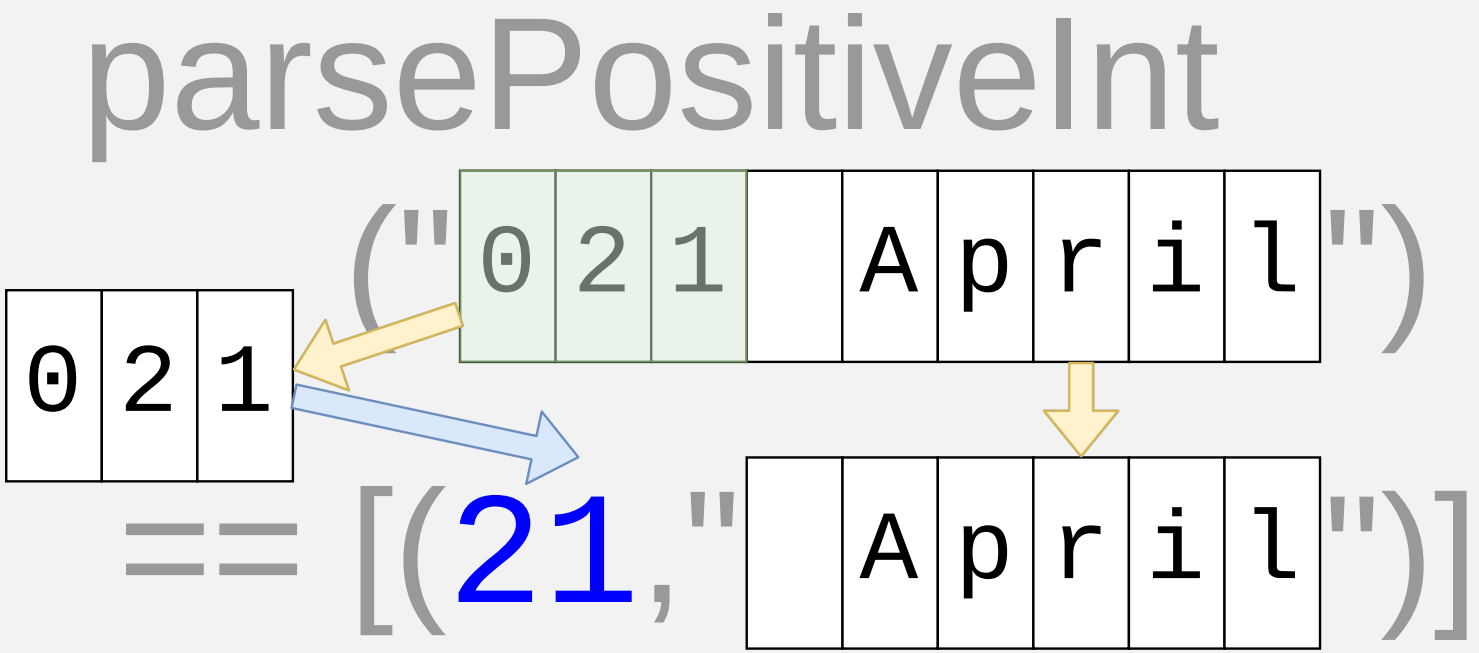
parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???,???)]
```



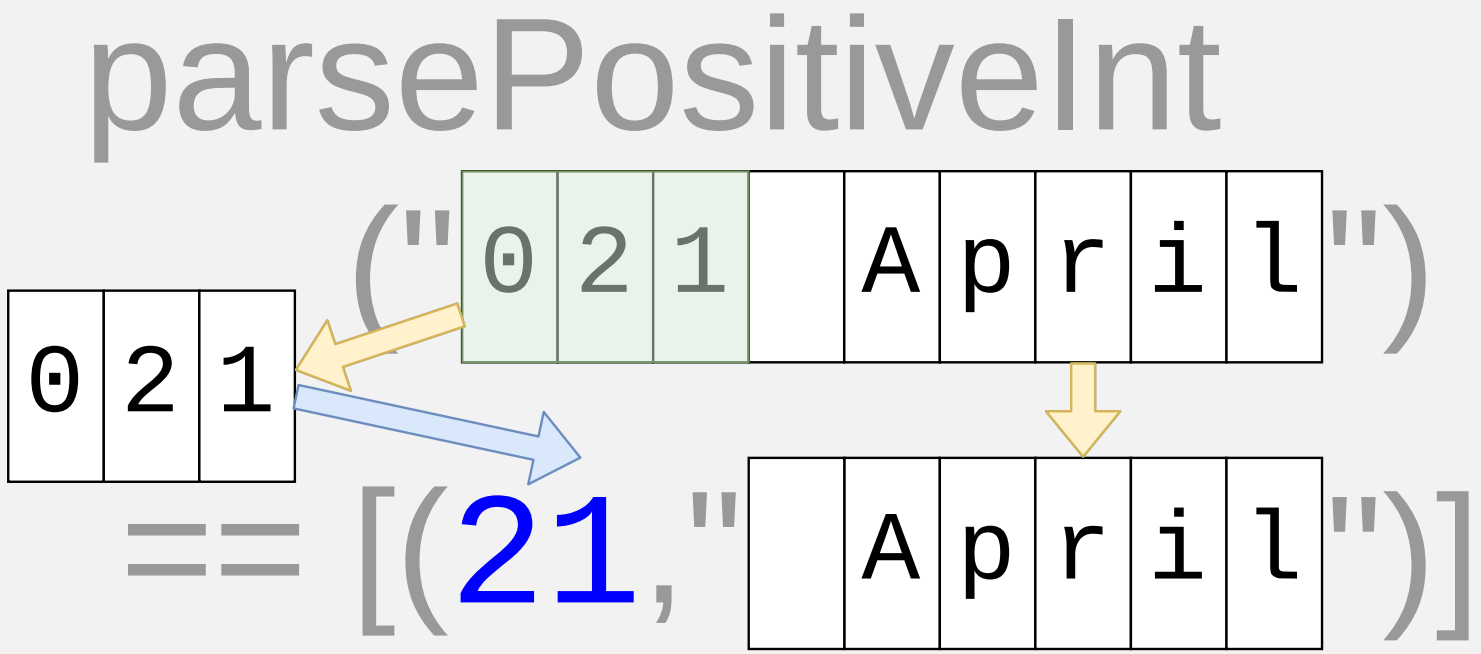
parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = ??? input
```



parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input
```



Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input
```

Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = ??? input
```

Aside: how to guess

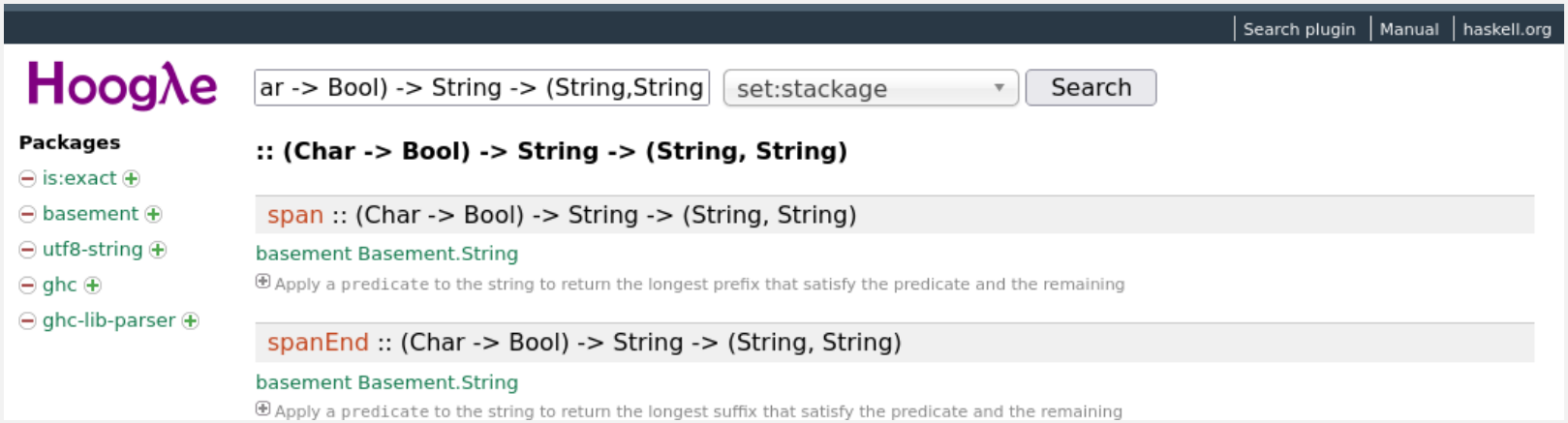
```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = ???0 input

???0 :: String -> (String, String)
```

Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = ???1 ???2 input

???1 :: (Char -> Bool) -> String -> (String, String)
???2 :: (Char -> Bool)
```

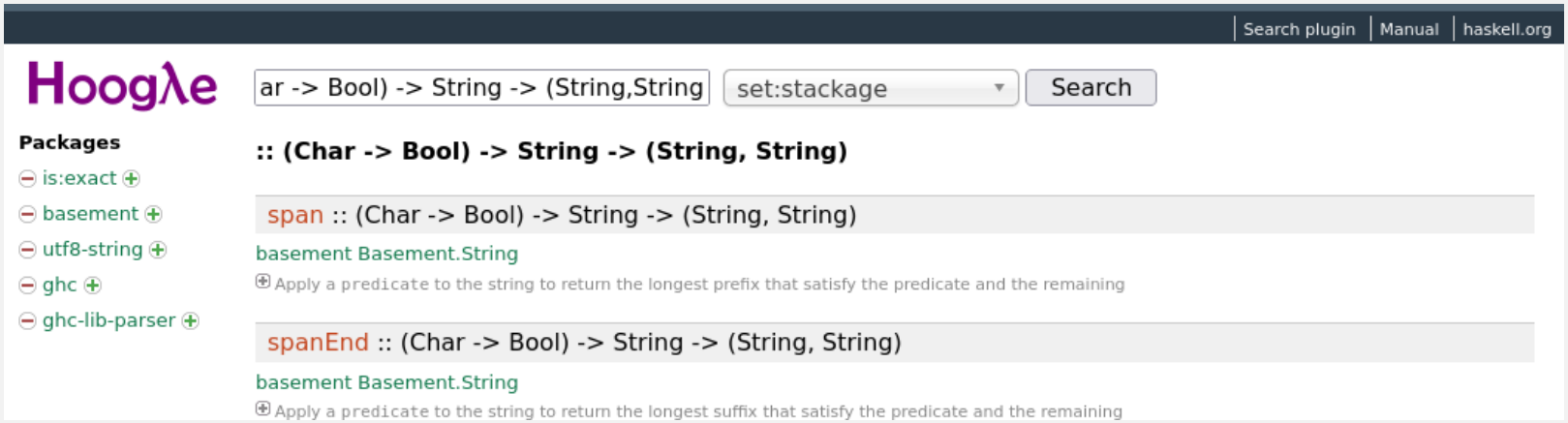
Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = ???1 ???2 input

???1 :: (Char -> Bool) -> String -> (String, String)
???2 :: (Char -> Bool)
```



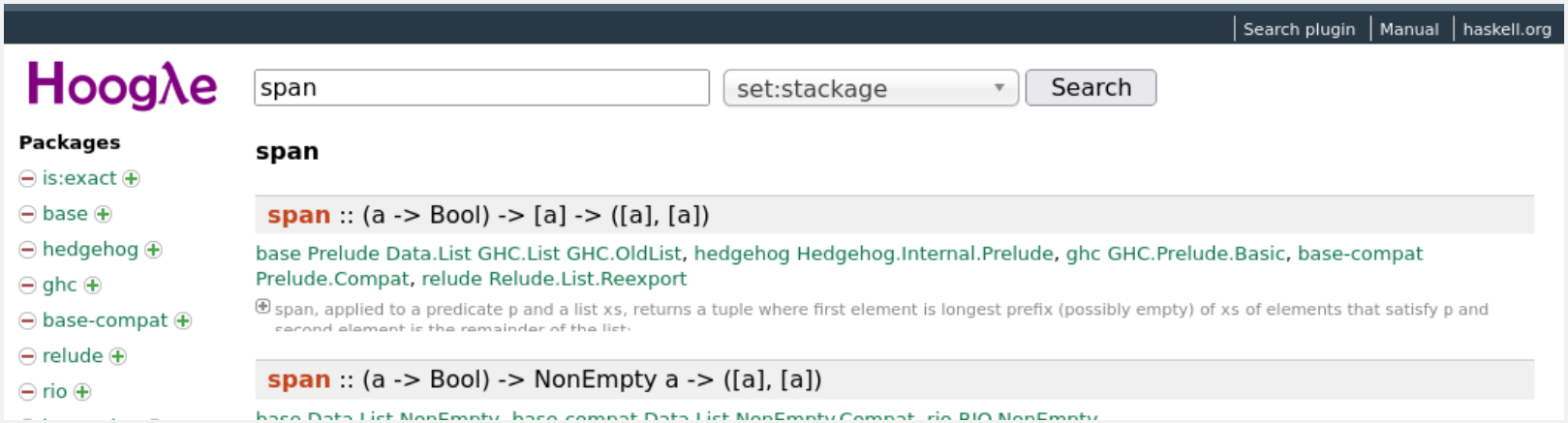
Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = Basement.String.span ???2 inp

Basement.String.span :: (Char -> Bool) -> String -> (S
???2 :: (Char -> Bool)
```



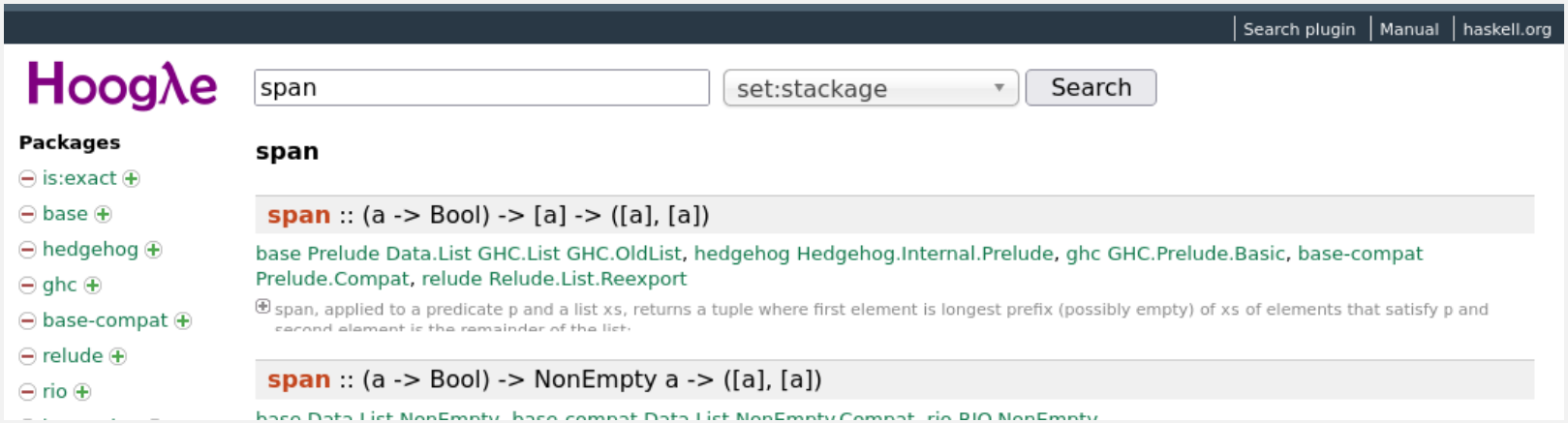
Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = Basement.String.span ???2 inp

Basement.String.span :: (Char -> Bool) -> String -> (S
???2 :: (Char -> Bool)
```



Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span ???2 input

span :: (a -> Bool) -> [a] -> ([a], [a])
???2 :: (Char -> Bool)
```

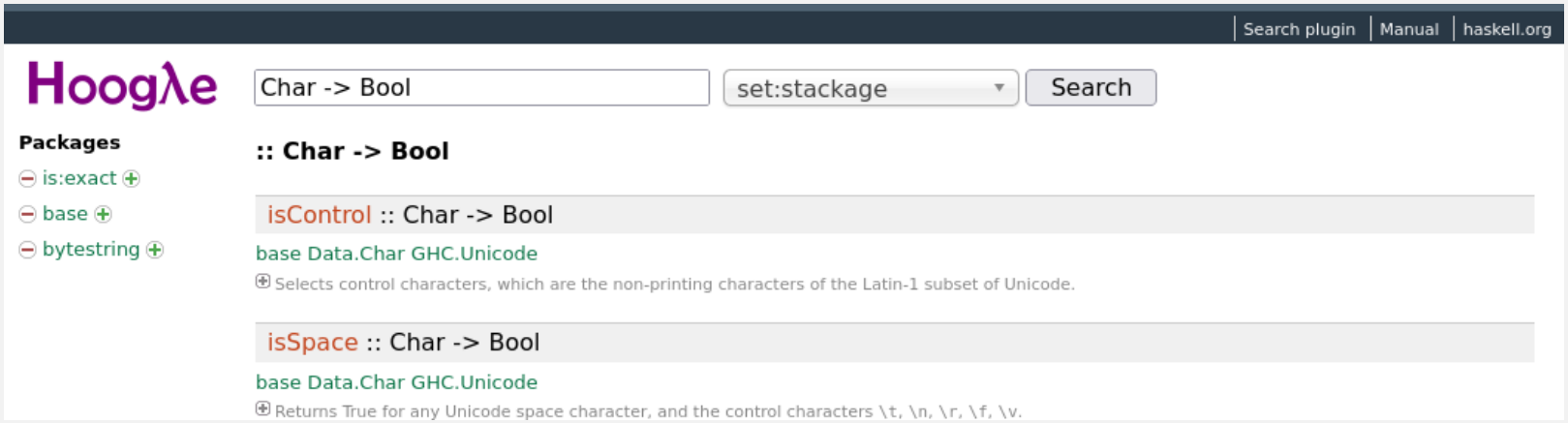


Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span ???2 input

span :: (a -> Bool) -> [a] -> ([a], [a])
???2 :: (Char -> Bool)
```

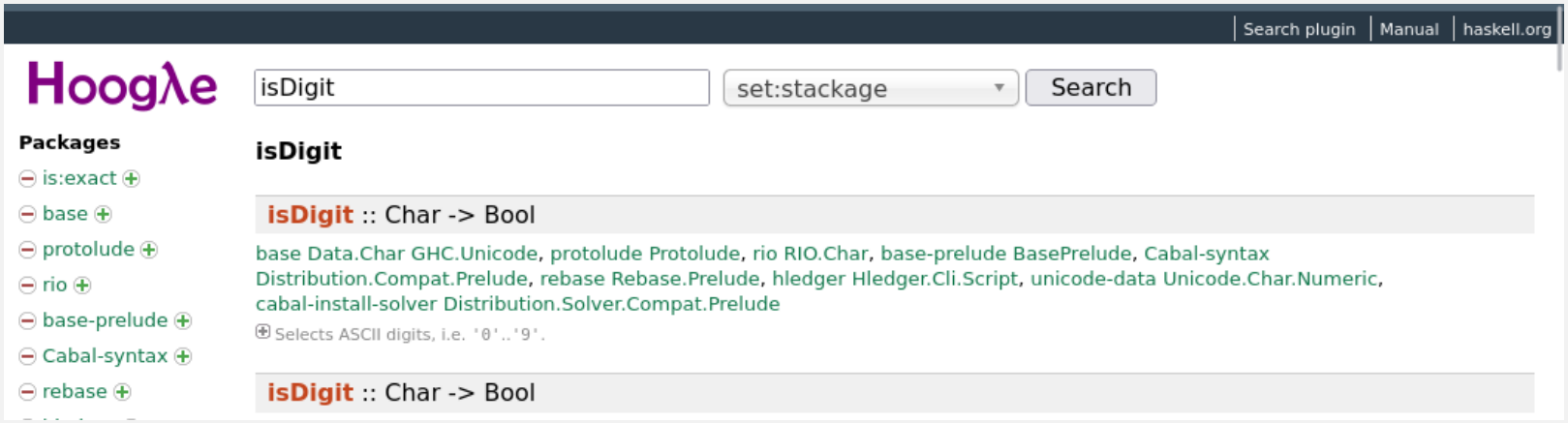
Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span ???2 input

span :: (a -> Bool) -> [a] -> ([a], [a])
???2 :: (Char -> Bool)
```



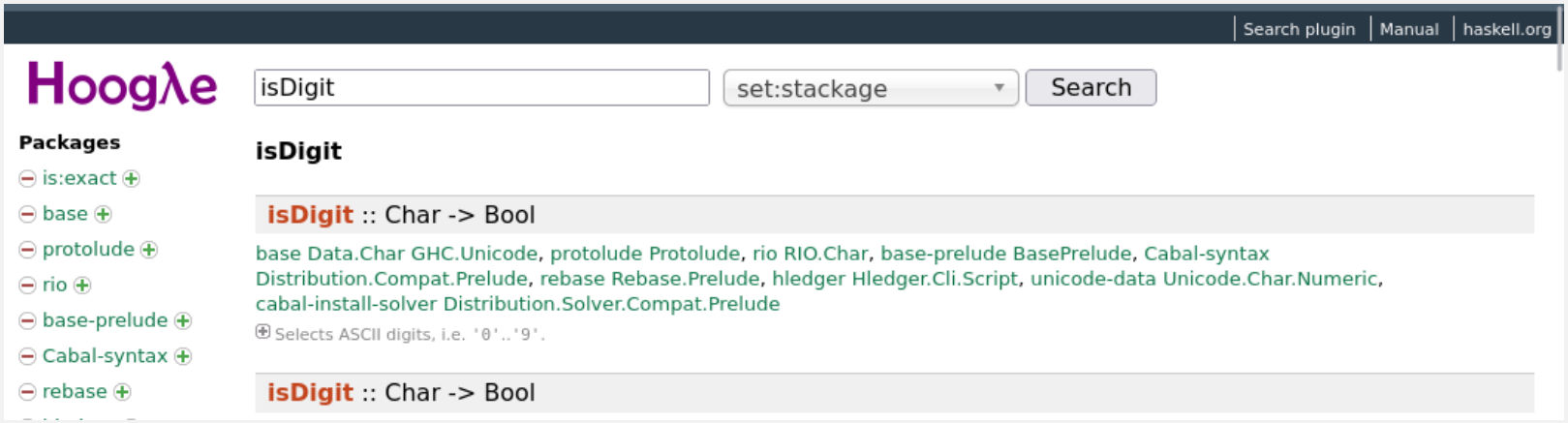
Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span ???2 input

span :: (a -> Bool) -> [a] -> ([a], [a])
???2 :: (Char -> Bool)
```



Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input

span :: (a -> Bool) -> [a] -> ([a], String)
isDigit :: (Char -> Bool)
```

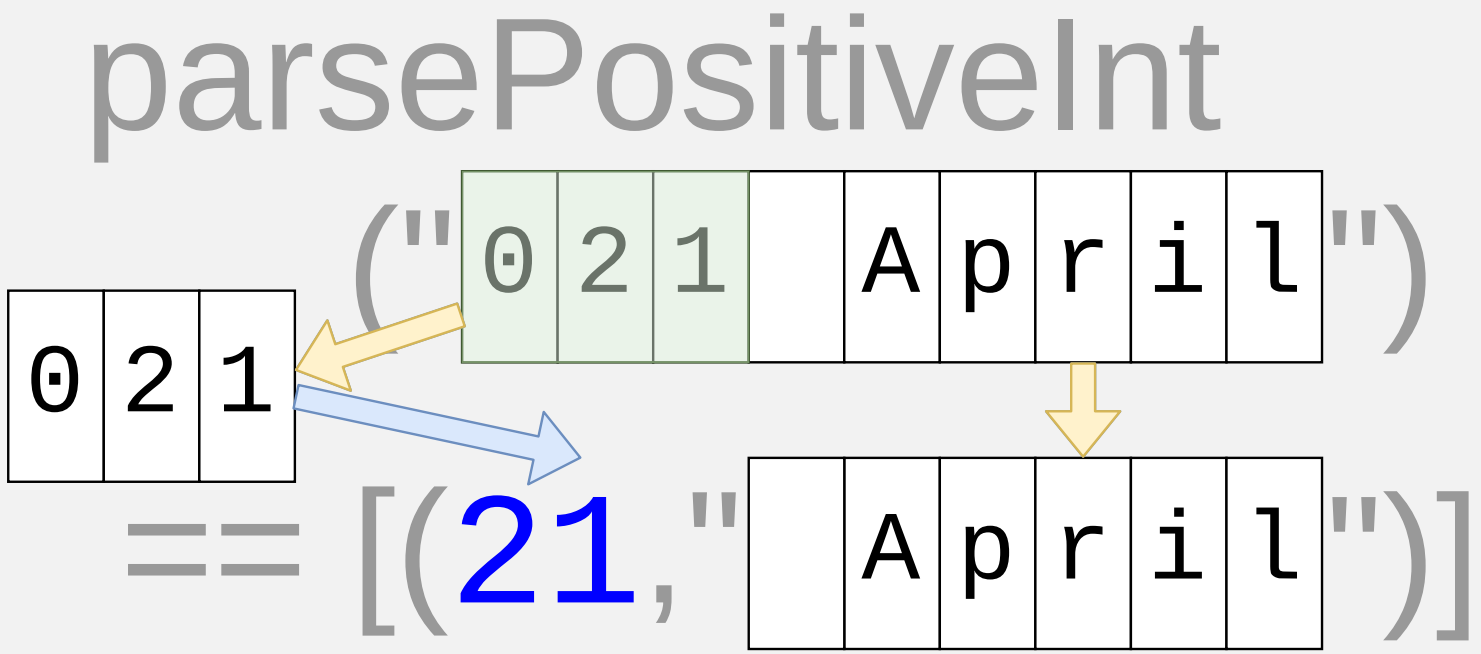


Aside: how to guess

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input

span :: (a -> Bool) -> [a] -> ([a], String)
isDigit :: (Char -> Bool)
```

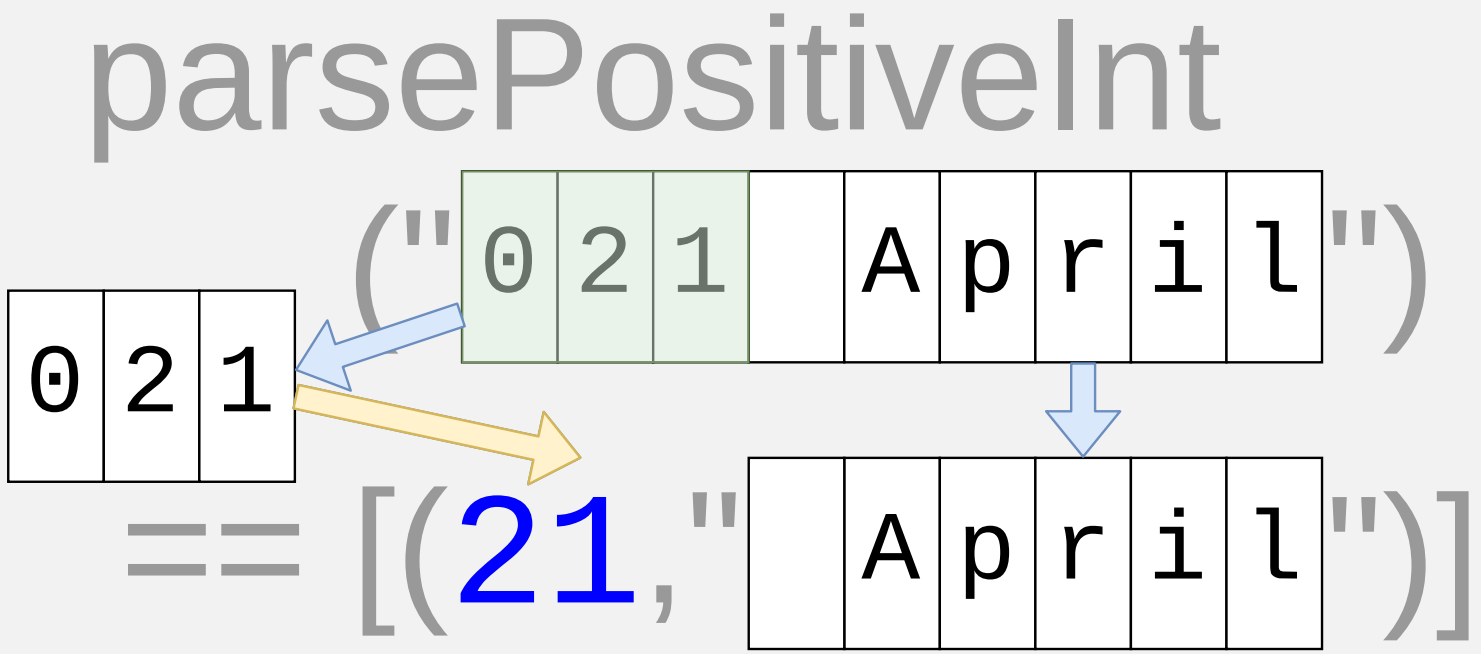
parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input
```



parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input
```



parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input

numStr :: String
??? :: Int
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(???, tail)]
  where (numStr, tail) = span isDigit input

numStr :: String
??? :: Int

read :: String -> Int -- built-in
```

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

numStr :: String
??? :: Int

read :: String -> Int -- built-in
```

parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]

parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

numStr :: String
??? :: Int

read :: (Read a) => String -> a -- built-in
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
read "Date {month = October, day = 31}" :: Date
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
read "Date {month = October, day = 31}" :: Date
 *** Exception: Prelude.read: no parse
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
read "Date {month = October, day = 31}" :: Date
 *** Exception: Prelude.read: no parse
read "Date 31 October" :: Date
 *** Exception: Prelude.read: no parse
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
read "Date {month = October, day = 31}" :: Date
 *** Exception: Prelude.read: no parse
read "Date 31 October" :: Date
 *** Exception: Prelude.read: no parse
read "31 Oct" :: Date
 *** Exception: Prelude.read: no parse
```

Aside: why not just `read` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day = 31, month = October}" :: Date
  == Date {day = 31, month = October}
read "Date {month = October, day = 31}" :: Date
 *** Exception: Prelude.read: no parse
read "Date 31 October" :: Date
 *** Exception: Prelude.read: no parse
read "31 Oct" :: Date
 *** Exception: Prelude.read: no parse
readMaybe :: (Read a) => String -> Maybe a -- built-in
```

Aside: why not just `readMaybe` everything?

```
data Date = Date {day :: Int, month :: Month}
  deriving (Read)
read "Date {day=31, month=October}" :: Date
  == Just (Date {day = 31, month = October})
readMaybe "Date {month=October, day=31}" :: Date
  == Nothing
readMaybe "Date 31 October" :: Date
  == Nothing
readMaybe "31 Oct" :: Date
  == Nothing
readMaybe :: (Read a) => String -> Maybe a -- built-in
```

Aside: why not just `readMaybe` everything?

🧪 Try it out!

Aside: why not just `readMaybe` everything?

- 🧪 Try it out!
- 💥 Beware its limits

Aside: why not just `readMaybe` everything?

🧪 Try it out!

💥 Beware its limits

Exercise

- Implement `readIntMaybe :: String -> Maybe Int`

Aside: why not just `readMaybe` everything?

🧪 Try it out!

💥 Beware its limits

Exercise

- Implement `readIntMaybe :: String -> Maybe Int`
- no `read` or `readMaybe`

Aside: why not just `readMaybe` everything?

🧪 Try it out!

💥 Beware its limits

Exercise

- Implement `readIntMaybe :: String -> Maybe Int`
- no `read` or `readMaybe`
- Hints:
  - First write `readCharMaybe :: Char -> Maybe Int`
  - `Char` can be pattern-matched

parsePositiveInt testing

```
parsePositiveInt :: Parser Int
parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

parsePositiveInt testing

```haskell
parsePositiveInt :: Parser Int
parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt "404" == [(404,"")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

`parsePositiveInt` testing

```haskell
parsePositiveInt :: Parser Int
parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

```haskell
parsePositiveInt "404" == [(404,"")] ✅
```

```haskell
parsePositiveInt "1 April" == [(1," April")] ✅
```

`parsePositiveInt` testing

```haskell
parsePositiveInt :: Parser Int
parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

```haskell
parsePositiveInt "404" == [(404,"")] ✅
```

```haskell
parsePositiveInt "1 April" == [(1," April")] ✅
```

```haskell
parsePositiveInt "007" == [(7,"")] ✅
```

`parsePositiveInt` testing

```haskell
parsePositiveInt :: Parser Int
parsePositiveInt input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

```haskell
parsePositiveInt "404" == [(404,"")] ✅
```

```haskell
parsePositiveInt "1 April" == [(1," April")] ✅
```

```haskell
parsePositiveInt "007" == [(7,"")] ✅
```

```haskell
parsePositiveInt "0" == [(0,"")] ❌
```

parsePositiveInt testing

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

```haskell
parsePositiveInt "404" == [(404,"")] ✅
```

```haskell
parsePositiveInt "1 April" == [(1," April")] ✅
```

```haskell
parsePositiveInt "007" == [(7,"")] ✅
```

```haskell
parsePositiveInt "0" == [(0,"")] ❌
```

parsePositiveInt testing

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

```haskell
parsePositiveInt "404" == [(404,"")] ✅
```

```haskell
parsePositiveInt "1 April" == [(1," April")] ✅
```

```haskell
parsePositiveInt "007" == [(7,"")] ✅
```

```haskell
parsePositiveInt "0" == [(0,"")] ❌
```

🧠

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard = ???
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser = ???
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = ???
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ ??? | ??? ]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ ???
  | ??? <- parser input ]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ ???
  | (result, tail) <- parser input ]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input ]
```

Hotfix: exclude 0

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input ]

(guard (> 0) parseNat) "0" == [(0,"")]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (???, tail)
  | (result, tail) <- parser input ]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result2, tail)
  | (result, tail) <- parser input
  , result2 <- if cond result then [result] else []]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input
  , cond result]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input
  , cond result]

[x | x <- [1..10], odd x] == [1,3,5,7,9]
```

Hotfix: exclude 0

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input
  , cond result]
```

Hotfix: exclude 0

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input
  , cond result]
```

`(guard (> 0) parseNat) "0" == []` ✅

Hotfix: exclude 0

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input

parsePositiveInt :: Parser Int
parsePositiveInt = guard (> 0) parseNat

guard :: (a -> Bool) -> Parser a -> Parser a
guard cond parser input = [ (result, tail)
  | (result, tail) <- parser input
  , cond result]

(guard (> 0) parseNat) "0" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅

parsePositiveInt "0" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅

parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number"
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅

parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number"

  *** Exception: Prelude.read: no parse
```

```
parsePositiveInt "this is not a number"
```

```
parsePositiveInt "this is not a number"
  == guard (> 0) parseNat "this is not a number"
```

```
parsePositiveInt "this is not a number"
  == guard (> 0) parseNat "this is not a number"
  == [ (result, tail) | (result, tail) <-
       parseNat "this is not a number"
     , (> 0) result]
```

```
parsePositiveInt "this is not a number"
  == guard (> 0) parseNat "this is not a number"
  == [ (result, tail) | (result, tail) <-
       parseNat "this is not a number"
     , (> 0) result]
  == [ (result, tail) | (result, tail) <-
       parseNat "this is not a number"
     , result > 0]
```

```
parsePositiveInt "this is not a number"
  == guard (> 0) parseNat "this is not a number"
  == [ (result, tail) | (result, tail) <-
       parseNat "this is not a number"
     , (> 0) result]
  == [ (result, tail) | (result, tail) <-
       parseNat "this is not a number"
     , result > 0]
  == [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = span isDigit "this is no
        in [(read numStr, tail)])
     , result > 0]
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

```
== [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = span isDigit "this is no
        in [(read numStr, tail)])
  , result > 0]
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

```
== [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = span isDigit "this is no
        in [(read numStr, tail)])
   , result > 0]

== [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = ("","this is not a numbe
        in [(read numStr, tail)])
   , result > 0]
```

```
== [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = span isDigit "this is no
        in [(read numStr, tail)])
   , result > 0]

== [ (result, tail) | (result, tail) <-
       (let (numStr, tail) = ("","this is not a numbe
        in [(read numStr, tail)])
   , result > 0]

== [ (result, tail) | (result, tail) <-
       (let  numStr = ""
             tail = "this is not a number"
        in [(read numStr, tail)])
   , result > 0]
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

```
== [ (result, tail) | (result, tail) <-
      (let  numStr = ""
            tail = "this is not a number"
       in [(read numStr, tail)])
  , result > 0]
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

```
== [ (result, tail) | (result, tail) <-
      (let  numStr = ""
            tail = "this is not a number"
       in [(read numStr, tail)])
  , result > 0]

== [ (result, tail) | (result, tail) <-
      (let  tail = "this is not a number"
       in [(read "", tail)])
  , result > 0]
```

```
   == [ (result, tail) | (result, tail) <-
         (let  numStr = ""
               tail = "this is not a number"
          in [(read numStr, tail)])
      , result > 0]

   == [ (result, tail) | (result, tail) <-
         (let  tail = "this is not a number"
          in [(read "", tail)])
      , result > 0]

*** Exception: Prelude.read: no parse
```

Fixing the crash

```
parseNat :: Parser Int
parseNat input = [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

Fixing the crash

```
parseNat :: Parser Int
parseNat input = if numStr=="" then [] else
                   [(read numStr, tail)]
  where (numStr, tail) = span isDigit input
```

Fixing the crash

```haskell
parseNat :: Parser Int
parseNat input = [(read numStr, tail) | numStr /= ""]
  where (numStr, tail) = span isDigit input
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅

parsePositiveInt "" == [] ✅
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅

parsePositiveInt "" == [] ✅

parsePositiveInt "1e10" == [("1","e10")] ❌
parsePositiveInt "0xB33F" == [("0","xB33F")] ❌
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅

parsePositiveInt "" == [] ✅

parsePositiveInt "1e10" == [("1","e10")] ❓
parsePositiveInt "0xB33F" == [("0","xB33F")] ❓
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅

parsePositiveInt "" == [] ✅

parsePositiveInt "1e10" == [("1","e10")] (✅)
parsePositiveInt "0xB33F" == [("0","xB33F")] (✅)
```

`parsePositiveInt` testing

```
parsePositiveInt "404" == [(404,"")] ✅
parsePositiveInt "1 April" == [(1," April")] ✅
parsePositiveInt "007" == [(7,"")] ✅
parsePositiveInt "0" == [] ✅

parsePositiveInt "this is not a number" == [] ✅

parsePositiveInt "-10" == [] ✅

parsePositiveInt "" == [] ✅

parsePositiveInt "1e10" == [("1","e10")] (✅)
parsePositiveInt "0xB33F" == [("0","xB33F")] (✅)
```

Progress on `parseDate`

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

Progress on `parseDate`

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

✅ `<$> :: (a -> b) -> Parser a -> Parser b`

✅ `<*> :: Parser (a -> b) -> Parser a -> Parser b`

✅ `parseDay :: Parser Int`

Progress on `parseDate`

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

✅ `<$> ::          (a -> b) -> Parser a -> Parser b`

✅ `<*> :: Parser (a -> b) -> Parser a -> Parser b`

✅ `parseDay :: Parser Int`

👉 `parseMonth :: Parser Month`

parseMonth

```
parseMonth :: Parser Month
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" ✅
parseMonth "august" ✅
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" ✅
parseMonth "august" ✅

parseMonth "nonsense" ❌
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" ✅
parseMonth "august" ✅

parseMonth "nonsense" ❌

parseMonth "jul" ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" ✅
parseMonth "august" ✅

parseMonth "nonsense" ❌

parseMonth "jul" ✅

parseMonth "j" ✅
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" ✅
parseMonth "august" ✅

parseMonth "nonsense" ❌

parseMonth "jul" ✅

parseMonth "j" ✅

parseMonth "" ❌
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" == [(July,"")]
parseMonth "august" == [(August,"")]

parseMonth "nonsense" == []

parseMonth "jul" == [(July,"")]

parseMonth "j" == [(January,""), (June,""), (July,"")]

parseMonth "" == []
```

parseMonth

```
parseMonth :: Parser Month

parseMonth "july" == [(July,"")]
```

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth "july" = [(July,"")]
```

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth "july" = [(July,"")]

parseMonth "july 2023"
*** Exception: <interactive>:3:41-71: Non-exhaustive p
```

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth ('j':'u':'l':'y':tail) = [(July,tail)]
```

Implementing `parseMonth`

```
parseMonth :: Parser Month

parseMonth ('j':'u':'l':'y':tail) = [(July,tail)]

parseMonth "july 2023" == [(July," 2023")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Implementing `parseMonth`

```
parseMonth :: Parser Month

parseMonth ('j':'u':'l':'y':tail) = [(July,tail)]

parseMonth "july 2023" == [(July," 2023")] ✅
```

But ugly!

🧠

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"
```

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"

parseString :: String -> Parser String
parseString = ???
```

Implementing parseString

```
parseString :: String -> Parser String
parseString str = ???
```

Implementing parseString

```
parseString :: String -> Parser String
parseString str input = ???
```

Implementing parseString

```
parseString :: String -> Parser String
parseString str input = ???
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

Implementing parseString

```
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  ??? -> ???
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

Implementing parseString

```
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> ???
  Just tail -> ???
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

Implementing parseString

```
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> ???
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

Implementing parseString

```haskell
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> [(str, tail)]
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

Testing parseString

```haskell
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> [(str, tail)]
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

```haskell
parseString "ja" "january" == [("ja", "nuary")] ✅
```

Testing parseString

```haskell
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> [(str, tail)]
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

```haskell
parseString "ja" "january" == [("ja", "nuary")] ✅

parseString "ja" "java" == [("ja", "va")] ✅
```

Testing parseString

```
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> [(str, tail)]
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list after the prefix, if it does.

```
parseString "ja" "january" == [("ja", "nuary")] ✅
parseString "ja" "java" == [("ja", "va")] ✅
parseString "ja" "nee" == [] ✅
```

Testing parseString

```haskell
parseString :: String -> Parser String
parseString str input = case stripPrefix str input of
  Nothing -> []
  Just tail -> [(str, tail)]
```

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

base Data.List GHC.OldList, rio RIO.List, base-prelude BasePrelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, LambdaHack Game.LambdaHack.Core.Prelude

The stripPrefix function drops the given prefix from a list. It returns Nothing if the list did not start with the prefix given, or Just the list

```haskell
parseString "ja" "january" == [("ja", "nuary")] ✅

parseString "ja" "java" == [("ja", "va")] ✅

parseString "ja" "nee" == [] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"

parseMonth "july" == [(July,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "jul" == [] ❌
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parseString "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "jul" == [] ❌
```

🧠

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String
```

Designing parsePrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String
```

Designing `parsePrefixOf`

```haskell
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String

parsePrefixOf "july" "july" ✅
```

Designing `parsePrefixOf`

```haskell
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String

parsePrefixOf "july" "july" ✅

parsePrefixOf "july" "jul" ✅
```

Designing `parsePrefixOf`

```haskell
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String

parsePrefixOf "july" "july" ✅

parsePrefixOf "july" "jul" ✅

parsePrefixOf "july" "ju" ✅
```

Designing `parsePrefixOf`

```
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String

parsePrefixOf "july" "july" ✅

parsePrefixOf "july" "jul" ✅

parsePrefixOf "july" "ju" ✅

parsePrefixOf "july" "j" ✅
```

Designing `parsePrefixOf`

```
parseMonth :: Parser Month

parseMonth = const July <$> parsePrefixOf "july"

parsePrefixOf :: String -> Parser String

parsePrefixOf "july" "july" ✅

parsePrefixOf "july" "jul" ✅

parsePrefixOf "july" "ju" ✅

parsePrefixOf "july" "j" ✅

parsePrefixOf "july" "" ❓
```

Designing `parsePrefixOf`

```
parseMonth :: Parser Month

parseMonth = const July <$> parseNEPrefixOf "july"

parseNEPrefixOf, parsePrefixOf :: String -> Parser Str

parseNEPrefixOf "july" "july" ✅

parseNEPrefixOf "july" "jul" ✅

parseNEPrefixOf "july" "ju" ✅

parseNEPrefixOf "july" "j" ✅

parseNEPrefixOf "july" "" ❌
```

Designing parsePrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parseNEPrefixOf "july"

parseNEPrefixOf, parsePrefixOf :: String -> Parser Str

parseNEPrefixOf "july" "july" ✅

parseNEPrefixOf "july" "jul" ✅

parseNEPrefixOf "july" "ju" ✅

parseNEPrefixOf "july" "j" ✅

parseNEPrefixOf "july" "" ❌

parseNEPrefixOf str = guard (/= "") (parsePrefixOf str
```

Designing parsePrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parseNEPrefixOf "july"

parseNEPrefixOf, parsePrefixOf :: String -> Parser Str

parsePrefixOf "july" "july" ✅

parsePrefixOf "july" "jul" ✅

parsePrefixOf "july" "ju" ✅

parsePrefixOf "july" "j" ✅

parsePrefixOf "july" "" ✅

parseNEPrefixOf str = guard (/= "") (parsePrefixOf str
```

Designing parsePrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parseNEPrefixOf "july"

parseNEPrefixOf, parsePrefixOf :: String -> Parser Str

parsePrefixOf "july" "july" == [("july","")]

parsePrefixOf "july" "jul" == [("jul","")]

parsePrefixOf "july" "ju" == [("ju","")]

parsePrefixOf "july" "j" == [("j","")]

parsePrefixOf "july" "" == [("","")]

parseNEPrefixOf str = guard (/= "") (parsePrefixOf str
```

Designing parsePrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parseNEPrefixOf "july"

parseNEPrefixOf, parsePrefixOf :: String -> Parser Str

parsePrefixOf "july" "july" =?= [("july",""),("jul","y

parsePrefixOf "july" "jul" =?= [("jul",""),("ju","l"),

parsePrefixOf "july" "ju" =?= [("ju",""),("j","u"),(""

parsePrefixOf "july" "j" =?= [("j",""),("","j")]

parsePrefixOf "july" "" =?= [("","")]

parseNEPrefixOf str = guard (/= "") (parsePrefixOf str
```

Designing parseLongestPrefixOf

```
parseMonth :: Parser Month

parseMonth = const July <$> parseLongestNEPrefixOf "ju

parseLongestNEPrefixOf, parseLongestPrefixOf :: String

parseLongestPrefixOf "july" "july" == [("july","")]

parseLongestPrefixOf "july" "jul" == [("jul","")]

parseLongestPrefixOf "july" "ju" == [("ju","")]

parseLongestPrefixOf "july" "j" == [("j","")]

parseLongestPrefixOf "july" "" == [("","")]

parseLongestNEPrefixOf str = guard (/= "") (parseLonge
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf = ???
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf str = ???
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf str input = ???
```

Implementing `parseLongestPrefixOf`

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js) = ???
parseLongestPrefixOf     []  input = ???
parseLongestPrefixOf    str     [] = ???
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js) = ???
parseLongestPrefixOf     []  input = [("",input)]
parseLongestPrefixOf    str     [] = ???
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js) = ???
parseLongestPrefixOf     []  input = [("",input)]
parseLongestPrefixOf    str     [] = [("","")]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js) = ???
parseLongestPrefixOf      _  input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js) =
  if i == j
  then ???
  else ???
parseLongestPrefixOf       _   input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j     = ???
  |  otherwise = ???
parseLongestPrefixOf       _   input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j     = ???
  |  otherwise = [("",j:js)]
parseLongestPrefixOf       _  input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ???
parseLongestPrefixOf       _   input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = (i:) <$> parseLongestPrefixOf is js
parseLongestPrefixOf       _  input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅
```

Implementing `parseLongestPrefixOf`

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅

parseLongestPrefixOf "bannana" "b" == [("b","")] ✅
```

Implementing `parseLongestPrefixOf`

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅

parseLongestPrefixOf "bannana" "b" == [("b","")] ✅

parseLongestPrefixOf "bannana" "" == [("","")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅

parseLongestPrefixOf "bannana" "b" == [("b","")] ✅

parseLongestPrefixOf "bannana" "" == [("","")] ✅

parseLongestPrefixOf "bannana" "bannanas"
  == [("bannana","s")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅

parseLongestPrefixOf "bannana" "b" == [("b","")] ✅

parseLongestPrefixOf "bannana" "" == [("","")] ✅

parseLongestPrefixOf "bannana" "bannanas"
  == [("bannana","s")] ✅

parseLongestPrefixOf "bannana" "banana"
  == [("ban","ana")] ✅
```

Implementing parseLongestPrefixOf

```
parseLongestPrefixOf :: String -> Parser String
parseLongestPrefixOf (i:is) (j:js)
  |  i == j = ((i:) <$> parseLongestPrefixOf is) js
parseLongestPrefixOf       _  input = [("",input)]

parseLongestPrefixOf "bannana" "ba" == [("ba","")] ✅

parseLongestPrefixOf "bannana" "b" == [("b","")] ✅

parseLongestPrefixOf "bannana" "" == [("","")] ✅

parseLongestPrefixOf "bannana" "bannanas"
  == [("bannana","s")] ✅

parseLongestPrefixOf "bannana" "banana"
  == [("ban","ana")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"

parseMonth "july" == [(July,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "july 2023" == [(July," 2023")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "july 2023" == [(July," 2023")] ✅

parseMonth "jul" == [(July,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "july 2023" == [(July," 2023")] ✅

parseMonth "jul" == [(July,"")] ✅

parseMonth "j" == [(July,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"
```

```
parseMonth "july" == [(July,"")] ✅

parseMonth "july 2023" == [(July," 2023")] ✅

parseMonth "jul" == [(July,"")] ✅

parseMonth "j" == [(July,"")] ✅

parseMonth "" == [] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const July <$> parseLongestNEPrefixOf "july"

parseMonth "july" == [(July,"")] ✅

parseMonth "july 2023" == [(July," 2023")] ✅

parseMonth "jul" == [(July,"")] ✅

parseMonth "j" == [(July,"")] ✅

parseMonth "" == [] ✅

parseMonth "august" == [] ❌
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const August <$> parseLongestNEPrefixOf "august"
```

```
parseMonth "july" == [(July,"")] ❌

parseMonth "july 2023" == [(July," 2023")] ❌

parseMonth "jul" == [(July,"")] ❌

parseMonth "j" == [(July,"")] ❌

parseMonth "" == [] ✅

parseMonth "august" == [] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const August <$> parseLongestNEPrefixOf "august"
```

```
parseMonth "july" == [(July,"")] ❌

parseMonth "july 2023" == [(July," 2023")] ❌

parseMonth "jul" == [(July,"")] ❌

parseMonth "j" == [(July,"")] ❌

parseMonth "" == [] ✅

parseMonth "august" == [] ✅
```

🧠

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const August <$> parseLongestNEPrefixOf "august"
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const August <$> parseLongestNEPrefixOf "august"
 <|> const July   <$> parseLongestNEPrefixOf "july"
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =   const August <$> parseLongestNEPrefixOf "august"
 <|> const July   <$> parseLongestNEPrefixOf "july"
 <|> const June   <$> parseLongestNEPrefixOf "june"
 <|> const May    <$> parseLongestNEPrefixOf "may"
 ...
```

Designing <|>

```
<|> :: Parser Month -> Parser Month -> Parser Month
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a
```

```
(parseString"bat"<|>parseString"bird") "bat" ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a
```

```
(parseString"bat"<|>parseString"bird") "bat" ✅
```

```
(parseString"bat"<|>parseString"bird") "bird" ✅
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a
```

```
(parseString"bat"<|>parseString"bird") "bat" ✅
```

```
(parseString"bat"<|>parseString"bird") "bird" ✅
```

```
(parseString"bat"<|>parseString"batman") "batman" ✅
```

Designing <|>

```
<|> :: Parser a -> Parser a
```

```
(parseString"bat"<|>parseString"bird") "bat" ✅
(parseString"bat"<|>parseString"bird") "bird" ✅
(parseString"bat"<|>parseString"batman") "batman" ✅
(parseString"bat"<|>parseString"batman") "bug" ❌
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")]

(parseString"bat"<|>parseString"bird") "bird" ✅

(parseString"bat"<|>parseString"batman") "batman" ✅

(parseString"bat"<|>parseString"batman") "bug" ❌
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")]

(parseString"bat"<|>parseString"bird") "bird"
  == [("bird","")]

(parseString"bat"<|>parseString"batman") "batman" ✅

(parseString"bat"<|>parseString"batman") "bug" ❌
```

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")]

(parseString"bat"<|>parseString"bird") "bird"
  == [("bird","")]

(parseString"bat"<|>parseString"batman") "batman"
  == [("bat","man"),("batman","")]

(parseString"bat"<|>parseString"batman") "bug"
```

❌

Designing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")]

(parseString"bat"<|>parseString"bird") "bird"
  == [("bird","")]

(parseString"bat"<|>parseString"batman") "batman"
  == [("bat","man"),("batman","")]

(parseString"bat"<|>parseString"batman") "bug"
  == []
```

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(<|>) = ???
```

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) = ???
```

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) input = ???
```

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) input = p1 input ++ p2 input
```

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) input = p1 input ++ p2 input
```

Exercise

- Is (++) the only way to combine `p1 input` with `p2 input`?
- What could go wrong?

Implementing <|>

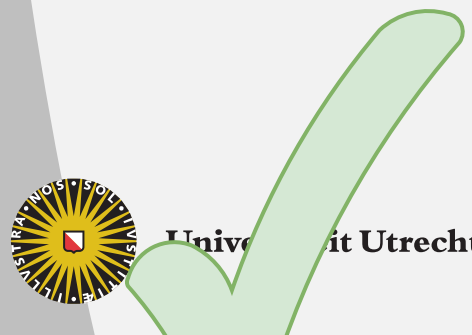
```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) input = p1 input ++ p2 input

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")] ✅

(parseString"bat"<|>parseString"bird") "bird"
  == [("bird","")] ✅

(parseString"bat"<|>parseString"batman") "batman"
  == [("bat","man"),("batman","")] ✅

(parseString"bat"<|>parseString"batman") "bug
  == [] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Implementing <|>

```
<|> :: Parser a -> Parser a -> Parser a

(p1 <|> p2) input = p1 input ++ p2 input

(parseString"bat"<|>parseString"bird") "bat"
  == [("bat","")] ✅

(parseString"bat"<|>parseString"bird") "bird"
  == [("bird","")] ✅

(parseString"bat"<|>parseString"batman") "batman"
  == [("bat","man"),("batman","")] ✅

(parseString"bat"<|>parseString"batman") "bug
  == [] ✅
```



Implementing `parseMonth`

```
parseMonth :: Parser Month
```

Implementing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const January   <$> parseLongestNEPrefixOf "janua
 <|> const February  <$> parseLongestNEPrefixOf "febru
 <|> const March     <$> parseLongestNEPrefixOf "march
 <|> const April     <$> parseLongestNEPrefixOf "april
 <|> const May       <$> parseLongestNEPrefixOf "may"
 <|> const June      <$> parseLongestNEPrefixOf "june"
 <|> const July      <$> parseLongestNEPrefixOf "july"
 <|> const August    <$> parseLongestNEPrefixOf "augus
 <|> const September <$> parseLongestNEPrefixOf "septe
 <|> const October   <$> parseLongestNEPrefixOf "octob
 <|> const November  <$> parseLongestNEPrefixOf "novem
 <|> const December  <$> parseLongestNEPrefixOf "decem
```

Testing `parseMonth`

```
parseMonth :: Parser Month
```

Testing `parseMonth`

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅
```

Testing `parseMonth`

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅
```

Testing `parseMonth`

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅
```

Testing `parseMonth`

```
parseMonth :: Parser Month
```

```
parseMonth "december" == [(December,"")] ✅
```

```
parseMonth "dec" == [(December,"")] ✅
```

```
parseMonth "december 2023" == [(December," 2023")] ✅
```

```
parseMonth "dec 2023" == [(December," 2023")] ✅
```

```
parseMonth "d 2023" == [(December," 2023")] ✅
```

Testing `parseMonth`

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅

parseMonth "d 2023" == [(December," 2023")] ✅

parseMonth "" == [] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅

parseMonth "d 2023" == [(December," 2023")] ✅

parseMonth "" == [] ✅

parseMonth "mar" == [(March,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅

parseMonth "d 2023" == [(December," 2023")] ✅

parseMonth "" == [] ✅

parseMonth "mar" == [(March,"")] ✅

parseMonth "j" == [(January,""),(June,""),(July,"")] ✅
```

Testing parseMonth

```
parseMonth :: Parser Month
```

```
parseMonth "december" == [(December,"")] ✅
```

```
parseMonth "dec" == [(December,"")] ✅
```

```
parseMonth "december 2023" == [(December," 2023")] ✅
```

```
parseMonth "dec 2023" == [(December," 2023")] ✅
```

```
parseMonth "d 2023" == [(December," 2023")] ✅
```

```
parseMonth "" == [] ✅
```

```
parseMonth "mar" == [(March,"")] ✅
```

```
parseMonth "j" == [(January,""),(June,""),(July,"")] ✅
```

Universiteit Utrecht

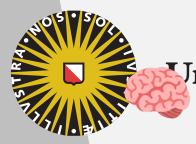
[Faculty of Science
Information and Computing
Sciences]

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅

parseMonth "d 2023" == [(December," 2023")] ✅

parseMonth "" == [] ✅

parseMonth "mar" == [(March,"")] ✅

parseMonth "j" == [(January,""),(June,""),(July,"")] ✅

parseMonth "june"
== [(January,"une"),(June,""),(July,"ne")] ❌
```

Testing parseMonth

```
parseMonth :: Parser Month

parseMonth "december" == [(December,"")] ✅

parseMonth "dec" == [(December,"")] ✅

parseMonth "december 2023" == [(December," 2023")] ✅

parseMonth "dec 2023" == [(December," 2023")] ✅

parseMonth "d 2023" == [(December," 2023")] ✅

parseMonth "" == [] ✅

parseMonth "mar" == [(March,"")] ✅

parseMonth "j" == [(January,""),(June,""),(July,"")] ✅

parseMonth "june"
== [(January,"une"),(June,""),(July,"ne")] ❌
```



Max. one month

```
longest :: Parser a -> Parser a

(longest parseMonth) "june" == [(June,"")]
```

Max. one month

```
longest :: Parser a -> Parser a

(longest parseMonth) "june" == [(June,"")]

(longest parseMonth) "ju" == [(June,""),(July,"")]
```

Max. one month

```
longest :: Parser a -> Parser a

(longest parseMonth) "june" == [(June,"")]

(longest parseMonth) "ju" == [(June,""),(July,"")]

(longest parseMonth) "april '99" == [(April," '99")]
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest = ???
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest parser = ???
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest parser input = ???
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest parser input = ???

  where outputs = parser input
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest parser input = ???

  where outputs = parser input

  -- e.g. [(January,"une"),(June,""),(July,"ne")]
```

Implementing `longest`

```
longest :: Parser a -> Parser a

longest parser input = ???

  where outputs = parser input

  -- e.g. [(January,"une"),(June,""),(July,"ne")]

       sorted = sortOn (length . snd) output
```

Implementing `longest`

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
        sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
        grouped = groupBy ((==) `on` length . snd) out
```

Implementing `longest`

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
        sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
        grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
```

Implementing longest

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
       sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
       grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
       oneGroup = take 1 grouped
```

Implementing longest

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
       sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
       grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
       oneGroup = take 1 grouped
  -- e.g. [[(June,"")]]
```

Implementing `longest`

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
        sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
        grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
        oneGroup = take 1 grouped
  -- e.g. [[(June,"")]]
       concatented = concat firstGroup
```

Implementing longest

```
longest :: Parser a -> Parser a
longest parser input = ???
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
        sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
        grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
        oneGroup = take 1 grouped
  -- e.g. [[(June,"")]]
       concatented = concat firstGroup
  -- e.g. [(June,"")]
```

Implementing longest

```
longest :: Parser a -> Parser a
longest parser input = concatented
  where outputs = parser input
  -- e.g. [(January,"une"),(June,""),(July,"ne")]
        sorted = sortOn (length . snd) output
  -- e.g. [(June,""),(July,"ne"),(January,"une")]
        grouped = groupBy ((==) `on` length . snd) out
  -- e.g. [[(June,"")],[(July,"ne")],[(January,"une")]
        oneGroup = take 1 grouped
  -- e.g. [[(June,"")]]
       concatented = concat firstGroup
  -- e.g. [(June,"")]
```

Tidying up longest

```
longest :: Parser a -> Parser a
longest parser input = concatented
  where outputs = parser input

        sorted = sortOn (length . snd) output

        grouped = groupBy ((==) `on` length . snd) out

        oneGroup = take 1 grouped

      concatented = concat firstGroup
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concatenated
  where outputs = parser input
        sorted = sortOn (length . snd) outputs
        grouped = groupBy ((==) `on` length . snd) sor
        firstGroup = take 1 grouped
        concatented = concat firstGroup
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concat
  $ firstGrouped
  where outputs = parser input
        sorted = sortOn (length . snd) outputs
        grouped = groupBy ((==) `on` length . snd) sor
        firstGroup = take 1 grouped
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concat
  . take 1
  $ grouped
  where outputs = parser input
        sorted = sortOn (length . snd) outputs
        grouped = groupBy ((==) `on` length . snd) sor
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  $ sorted
  where outputs = parser input
        sorted = sortOn (length . snd) outputs
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  $ outputs
  where outputs = parser input
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser input
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
  $ input
```

Tidying up longest

```
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
```

Tidying up `longest`

```haskell
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
```

⚠️⏱️ Performance alert! ⏱️⚠️

- `longest` is slow
- Exercise: speed it up algorithmically

Tidying up `longest`

```haskell
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
```

⚠️⏱️ Performance alert! ⏱️⚠️

- `longest` is slow
- Exercise: speed it up algorithmically
  - Hint 1: you can change the `Parser` type

Tidying up `longest`

```
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
```

⚠️⏱️ Performance alert! ⏱️⚠️

- `longest` is slow
- Exercise: speed it up algorithmically
  - Hint 1: you can change the `Parser` type
  - Hint 2: `length` is the biggest culprit

Tidying up `longest`

```haskell
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser
```

⚠️⏱️ Performance alert! ⏱️⚠️

- `longest` is slow
- Exercise: speed it up algorithmically
  - Hint 1: you can change the `Parser` type
  - Hint 2: `length` is the biggest culprit
  - Hint 3: why lists?

Testing longest

```haskell
longest :: Parser a -> Parser a

longest parser
  = concat
  . take 1
  . groupBy ((==) `on` length . snd)
  . sortOn (length . snd)
  . parser

longest (parseString "bat" | parseString "batman")
  == [("batman","")]  ✅
```

Fixing parseMonth

```
parseMonth :: Parser Month

parseMonth
  =  const January   <$> parseLongestNEPrefixOf "janua
 <|> const February  <$> parseLongestNEPrefixOf "febru
 <|> const March     <$> parseLongestNEPrefixOf "march
 <|> const April     <$> parseLongestNEPrefixOf "april
 <|> const May       <$> parseLongestNEPrefixOf "may"
 <|> const June      <$> parseLongestNEPrefixOf "june"
 <|> const July      <$> parseLongestNEPrefixOf "july"
 <|> const August    <$> parseLongestNEPrefixOf "augus
 <|> const September <$> parseLongestNEPrefixOf "septe
 <|> const October   <$> parseLongestNEPrefixOf "octob
 <|> const November  <$> parseLongestNEPrefixOf "novem
 <|> const December  <$> parseLongestNEPrefixOf "decem

parseMonth "june"
== [(January,"une"),(June,""),(July,"ne")]
```

Fixing parseMonth

```
parseMonth :: Parser Month
parseMonth = longest
  $  const January   <$> parseLongestNEPrefixOf "janua
 <|> const February  <$> parseLongestNEPrefixOf "febru
 <|> const March     <$> parseLongestNEPrefixOf "march
 <|> const April     <$> parseLongestNEPrefixOf "april
 <|> const May       <$> parseLongestNEPrefixOf "may"
 <|> const June      <$> parseLongestNEPrefixOf "june"
 <|> const July      <$> parseLongestNEPrefixOf "july"
 <|> const August    <$> parseLongestNEPrefixOf "augus
 <|> const September <$> parseLongestNEPrefixOf "septe
 <|> const October   <$> parseLongestNEPrefixOf "octob
 <|> const November  <$> parseLongestNEPrefixOf "novem
 <|> const December  <$> parseLongestNEPrefixOf "decem

parseMonth "june" == [(June,"")] ✅
```

Tidying up parseMonth

```
parseMonth :: Parser Month
parseMonth = longest
  $  January   <$ parseLongestNEPrefixOf "january"
 <|> February  <$ parseLongestNEPrefixOf "february"
 <|> March     <$ parseLongestNEPrefixOf "march"
 <|> April     <$ parseLongestNEPrefixOf "april"
 <|> May       <$ parseLongestNEPrefixOf "may"
 <|> June      <$ parseLongestNEPrefixOf "june"
 <|> July      <$ parseLongestNEPrefixOf "july"
 <|> August    <$ parseLongestNEPrefixOf "august"
 <|> September <$ parseLongestNEPrefixOf "september"
 <|> October   <$ parseLongestNEPrefixOf "october"
 <|> November  <$ parseLongestNEPrefixOf "november"
 <|> December  <$ parseLongestNEPrefixOf "december"
```

Tidying up parseMonth

```
parseMonth :: Parser Month
parseMonth = longest
  $  January   <$ parseLongestNEPrefixOf "january"
 <|> February  <$ parseLongestNEPrefixOf "february"
 <|> March     <$ parseLongestNEPrefixOf "march"
 <|> April     <$ parseLongestNEPrefixOf "april"
 <|> May       <$ parseLongestNEPrefixOf "may"
 <|> June      <$ parseLongestNEPrefixOf "june"
 <|> July      <$ parseLongestNEPrefixOf "july"
 <|> August    <$ parseLongestNEPrefixOf "august"
 <|> September <$ parseLongestNEPrefixOf "september"
 <|> October   <$ parseLongestNEPrefixOf "october"
 <|> November  <$ parseLongestNEPrefixOf "november"
 <|> December  <$ parseLongestNEPrefixOf "december"

<$ :: a -> Parser b -> Parser a
(x <$ p) = const x <$> p
```

Tidying up parseMonth

```
parseMonth :: Parser Month
parseMonth = longest
  $  January   <$ parseLongestNEPrefixOf "january"
 <|> February  <$ parseLongestNEPrefixOf "february"
 <|> March     <$ parseLongestNEPrefixOf "march"
 <|> April     <$ parseLongestNEPrefixOf "april"
 <|> May       <$ parseLongestNEPrefixOf "may"
 <|> June      <$ parseLongestNEPrefixOf "june"
 <|> July      <$ parseLongestNEPrefixOf "july"
 <|> August    <$ parseLongestNEPrefixOf "august"
 <|> September <$ parseLongestNEPrefixOf "september"
 <|> October   <$ parseLongestNEPrefixOf "october"
 <|> November  <$ parseLongestNEPrefixOf "november"
 <|> December  <$ parseLongestNEPrefixOf "december"
```

Progress on `parseDate`

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

Progress on `parseDate`

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

✅ `<$> ::          (a -> b) -> Parser a -> Parser b`

✅ `<*> :: Parser (a -> b) -> Parser a -> Parser b`

✅ `parseDay :: Parser Int`

✅ `parseMonth :: Parser Month`

Testing parseDate

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

parseDate6 "31 oct"
```

Testing parseDate

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

parseDate6 "31 oct"
  == [] ❌
```

Testing parseDate

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

parseDate6 "31 oct"
  == [] ❌

parseDate6 "31oct"
  == [(Date {day = 31, month = October},"")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Testing parseDate

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth

parseDate6 "31 oct"
  == [] ❌

parseDate6 "31oct"
  == [(Date {day = 31, month = October},"")] ✅
```

🧠

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Handling Whitespace

```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth
```

Handling Whitespace

```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth

parseSpaces :: Parser String
<* :: Parser a -> Parser b -> Parser a
```

Handling Whitespace

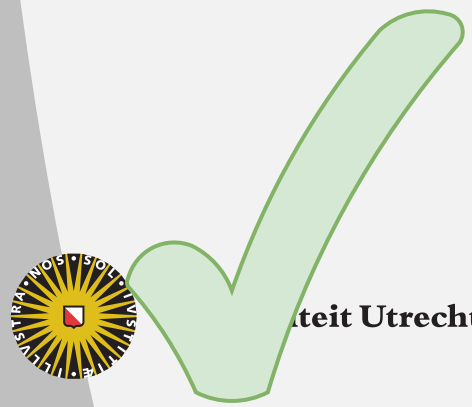
```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth

parseSpaces :: Parser String
<* :: Parser a -> Parser b -> Parser a

parseDate6 "31 oct"
  == [(Date 31 October,"")] ✅
```

Handling Whitespace

```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth

parseSpaces :: Parser String
<* :: Parser a -> Parser b -> Parser a

parseDate6 "31 oct"
  == [(Date 31 October,"")] ✅

parseDate6 "4 ju"
  == [ (Date 4 June,"")
     , (Date 4 July,"")] ✅
```

Handling Whitespace

```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth

parseSpaces :: Parser String
<* :: Parser a -> Parser b -> Parser a

parseDate6 "31 oct"
  == [(Date 31 October,"")] ✅

parseDate6 "4 ju"
  == [ (Date 4 June,"")
     , (Date 4 July,"")] ✅
```



Handling Whitespace

```
parseDate7 :: Parser Date
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth
```

Exercises

- `parseSpaces :: Parser String`
- `<* :: Parser a -> Parser b -> Parser a`

Handling Whitespace another way

```
type Parser a = String -> [(a,String)]
```

Handling Whitespace another way

```
type Parser a = [Char] -> [(a,[Char])]
```

Handling Whitespace another way

```
type Parser' tok a =  [tok] -> [(a, [tok])]
```

Handling Whitespace another way

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a
```

Handling Whitespace another way

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a

<*> :: Parser' tok (a -> b)
    -> Parser' tok  a
    -> Parser' tok       b
<$> ::                (a -> b)
    -> Parser' tok  a
    -> Parser' tok       b
...
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Building high-level parsers

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a
```

Building high-level parsers

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a

parseDate8 :: Parser' String Month
parseDate8 = Date <$> tok parseDay <*> tok parseMonth
```

Building high-level parsers

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a

parseDate8 :: Parser' String Month
parseDate8 = Date <$> tok parseDay <*> tok parseMonth

parseDay   :: Parser Int     -- unchanged
parseMonth :: Parser Month   -- unchanged
```

Building high-level parsers

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a

parseDate8 :: Parser' String Month
parseDate8 = Date <$> tok parseDay <*> tok parseMonth

parseDay   :: Parser Int     -- unchanged
parseMonth :: Parser Month   -- unchanged

tok :: Parser a -> Parser' String a
```

Building high-level parsers

```
type Parser' tok a =  [tok] -> [(a, [tok])]

type Parser a = Parser' Char a

parseDate8 :: Parser' String Month
parseDate8 = Date <$> tok parseDay <*> tok parseMonth

parseDay   :: Parser Int     -- unchanged
parseMonth :: Parser Month   -- unchanged
```

Exercise:

```
-- Parse a single token
tok :: Parser' subToken a
    -> Parser' [subToken] a
```

Running high-level parsers

```
parseDate8 ["31","oct"]
  == [(Date {day = 31, month = October},"")]
```

Running high-level parsers

```
parseDate8 (words "31 oct")
  == [(Date {day = 31, month = October},"")]
```

Running high-level parsers

```
parseDate8 (words "31 oct")
  == [(Date {day = 31, month = October},"")]

words :: String -> [String] -- built-in
```

Running high-level parsers

```
parseDate8 (words "31 oct")
  == [(Date {day = 31, month = October},"")]

words :: String -> [String] -- built-in

lines :: String -> [String] -- built-in

split :: Char -> String -> [String] -- built-in
```

Running high-level parsers

```
parseDate8 (words "31 oct")
  == [(Date {day = 31, month = October},"")]

words :: String -> [String] -- built-in

lines :: String -> [String] -- built-in

split :: Char -> String -> [String] -- built-in

removeCommentsAndWhitespace :: String -> [String]
```

Why bother?

✨🧹 high-level parsers

```
parseDate8 =
  Date <$> tok parseDay <*> tok parseMonth
```

vs.

```
parseDate7 =
  Date <$> parseDay <* parseSpaces <*> parseMonth
```

✨🧹 low-level parsers

```
parseNat input = [(read numStr, tail) | numStr /= ""]
  where (numStr, tail) = span isDigit input
```

vs.

```
parseNat input = [ (x,"") |
  [Just x] <- readMaybe input, x >= 0]
```

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month
```

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month

<$  ::              a -> Parser b -> Parser a
<*  ::       Parser a -> Parser b -> Parser a
<|> ::       Parser a -> Parser a -> Parser a
```

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month

<$  ::                a -> Parser b -> Parser a
<*  ::          Parser a -> Parser b -> Parser a
<|> ::          Parser a -> Parser a -> Parser a

parseDate7 :: Parser Date
```

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month

<$  ::                a -> Parser b -> Parser a
<*  ::          Parser a -> Parser b -> Parser a
<|> ::          Parser a -> Parser a -> Parser a

parseDate7 :: Parser Date

parseDate7 "31 oct"
  == [(Date 31 October,"")] ✅
```

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month

<$  ::                a -> Parser b -> Parser a
<*  ::          Parser a -> Parser b -> Parser a
<|> ::          Parser a -> Parser a -> Parser a

parseDate7 :: Parser Date

parseDate7 "31 oct"
  == [(Date 31 October,"")] ✅

parseDate7 "4 ju"
  == [ (Date 4 June,"")
     , (Date 4 July,"")] ✅
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Summary

```
parseDay :: Parser Int
parseMonth :: Parser Month

<$  ::                a -> Parser b -> Parser a
<*  ::          Parser a -> Parser b -> Parser a
<|> ::          Parser a -> Parser a -> Parser a

parseDate7 :: Parser Date

parseDate7 "31 oct"
  == [(Date 31 October,"")] ✅

parseDate7 "4 ju"
  == [ (Date 4 June,"")
     , (Date 4 July,"")] ✅

type Parser' tok a = [tok] -> [(a,[tok])]
```

More Exercises

- Support capital case anywhere:
  ✅ `Apr`, ✅ `apri`, ✅ `APRIL`, ✅ `ApRiL`
- Support capital case for first letter only:
  ✅ `Apr`, ✅ `apri`, ❌ `APRIL`, ❌ `ApRiL`