

Lecture 3: Parser Combinators

Talen en Compilers 2023-2024, period 2

Lawrence Chonavel

Department of Information and Computing Sciences, Utrecht University



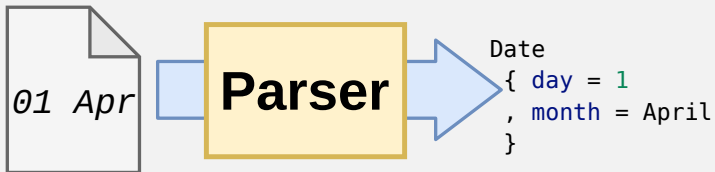
Recap: Parsers



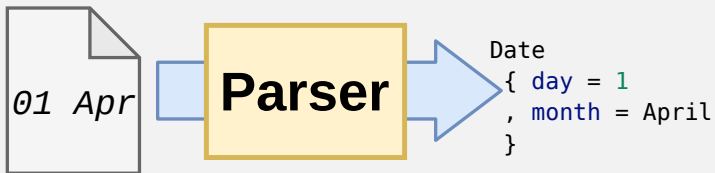
Recap: Parsers



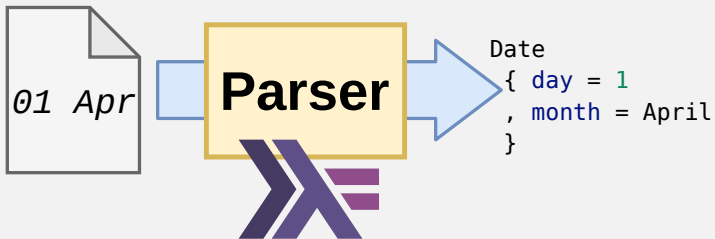
Recap: Parsers



Today's lecture

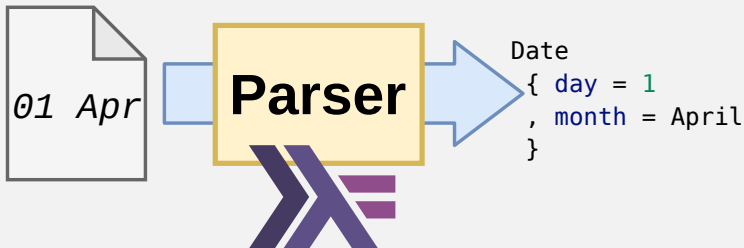


Today's lecture



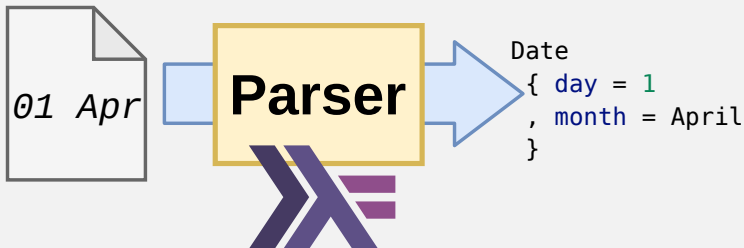
Attempt 1

```
parseDate :: ???
```



Attempt 1

```
parseDate1 :: String -> Date
```



Attempt 1

```
parseDate1 :: String -> Date  
  
parseDate1 "01 Apr" == Date {day = 1, month = April}  
parseDate1 "5 Nov" == Date {day = 5, month = November}  
parseDate1 "25 Dec" == Date {day = 25, month = December}
```



Attempt 1

```
parseDate1 :: String -> Date
```

```
parseDate1 "01 Apr" == Date {day = 1, month = April}
```

```
parseDate1 "5 Nov" == Date {day = 5, month = November}
```

```
parseDate1 "25 Dec" == Date {day = 25, month = December}
```

```
parseDate1 "007 Bond" == ???
```

```
parseDate1 "youtu.be/dQw4w9WgXcQ" == ???
```



Attempt 1

```
parseDate1 :: String -> Date  
  
parseDate1 "01 Apr" == Date {day = 1, month = April}  
parseDate1 "5 Nov" == Date {day = 5, month = November}  
parseDate1 "25 Dec" == Date {day = 25, month = December}  
  
parseDate1 "007 Bond" == ???  
parseDate1 "youtu.be/dQw4w9WgXcQ" == ???  
  
data Maybe a = Just a | Nothing -- built in
```



Attempt 2

```
parseDate2 :: String -> Maybe Date
```

```
parseDate2 "01 Apr" == Just (Date {day = 1, month = Apr})
```

```
parseDate2 "5 Nov" == Just (Date {day = 5, month = Nov})
```

```
parseDate2 "25 Dec" == Just (Date {day = 25, month = Dec})
```

```
parseDate2 "007 Bond" == Nothing
```

```
parseDate2 "youtu.be/dQw4w9WgXcQ" == Nothing
```

```
data Maybe a = Just a | Nothing -- built in
```



Attempt 2

```
parseDate2 :: String -> Maybe Date
```

```
parseDate2 "01 Apr" == Just (Date {day = 1, month = Ap
```

```
parseDate2 "5 Nov" == Just (Date {day = 5, month = Nov
```

```
parseDate2 "25 Dec" == Just (Date {day = 25, month = D
```

```
parseDate2 "007 Bond" == Nothing
```

```
parseDate2 "youtu.be/dQw4w9WgXcQ" == Nothing
```



Attempt 2

```
parseDate2 :: String -> Maybe Date
```

```
parseDate2 "01 Apr" == Just (Date {day = 1, month = Ap
```

```
parseDate2 "5 Nov" == Just (Date {day = 5, month = Nov
```

```
parseDate2 "25 Dec" == Just (Date {day = 25, month = D
```

```
parseDate2 "007 Bond" == Nothing
```

```
parseDate2 "youtu.be/dQw4w9WgXcQ" == Nothing
```

```
parseDate2 "4 Ju"
```

```
=?= Just (Date {day = 4, month = June})
```

```
=?= Just (Date {day = 4, month = July})
```



Attempt 3

```
parseDate3 :: String -> [Date]

parseDate3 "01 Apr" == [Date {day = 1, month = April}]
parseDate3 "5 Nov" == [Date {day = 5, month = November}]
parseDate3 "25 Dec" == [Date {day = 25, month = December}]

parseDate3 "007 Bond" == []
parseDate3 "youtu.be/dQw4w9WgXcQ" == []

parseDate3 "4 Ju" ==
    [Date {day = 4, month = June}
    ,Date {day = 4, month = July}]
```



Attempt 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 "01 Apr" == [Date {day = 1, month = April}]
```

```
parseDate3 "5 Nov" == [Date {day = 5, month = November}]
```

```
parseDate3 "25 Dec" == [Date {day = 25, month = December}]
```

```
parseDate3 "007 Bond" == []
```

```
parseDate3 "youtu.be/dQw4w9WgXcQ" == []
```

```
parseDate3 "4 Ju" ==
```

```
    [Date {day = 4, month = June}
```

```
    ,Date {day = 4, month = July}]
```



Attempt 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 "01 Apr" == [Date {day = 1, month = April}]
```

```
parseDate3 "5 Nov" == [Date {day = 5, month = November}]
```

```
parseDate3 "25 Dec" == [Date {day = 25, month = December}]
```

```
parseDate3 "007 Bond" == []
```

```
parseDate3 "youtu.be/dQw4w9WgXcQ" == []
```

```
parseDate3 "4 Ju" ==
```

```
    [Date {day = 4, month = June}
```

```
    ,Date {day = 4, month = July}]
```



Implementing 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 input = ???
```



Implementing 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 input = ???
```



Implementing 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 input = ???
```

```
parseDay3 :: String -> [Int]
```

```
parseMonth3 :: String -> [Month]
```



Implementing 3

```
parseDate3 :: String -> [Date]
```

```
parseDate3 input = [ ???  
  | d <- parseDay3 ???  
  , m <- parseMonth3 ???]
```



Aside: Haskell list comprehensions

```
[ x * 2 | x <- [1,2,3] ]  
== [2,4,6]
```



Aside: Haskell list comprehensions

```
[ x * 2 | x <- [1,2,3] ]
```

```
== [2,4,6]
```

```
[ x * y | x <- [1,2,3]  
          , y <- [1,10,100]]
```

```
== [1,10,100,2,20,200,3,30,300]
```



Aside: Haskell list comprehensions

```
[ x * 2 | x <- [1,2,3] ]
```

```
== [2,4,6]
```

```
[ x * y | x <- [1,2,3]  
        , y <- [1,10,100]]
```

```
== [1,10,100,2,20,200,3,30,300]
```

```
[x * y | x <- [1,2,3]  
        , y <- filter (>= x) [1,2,3]]
```

```
== [1,2,3,4,6,9]
```



Back to Implementing 3

```
parseDate3 :: String -> [Date]
parseDate3 input = [ ???
  | d <- parseDay3 ???
  , m <- parseMonth3 ???]
```



Back to Implementing 3

```
parseDate3 :: String -> [Date]
parseDate3 input = [ Date {day=d, month=m}
  | d <- parseDay3 ???
  , m <- parseMonth3 ???]
```



Back to Implementing 3

```
parseDate3 :: String -> [Date]
parseDate3 input = [ Date d m
  | d <- parseDay3 ???
  , m <- parseMonth3 ???]
```



How to split input?

```
parseDate3 :: String -> [Date]
parseDate3 input = [ Date d m
  | d <- parseDay3 ???{- begining of s -}
  , m <- parseMonth3 ???{- end of s -}]
```



How to split input?

```
parseDate3 :: String -> [Date]
parseDate3 input = [ Date d m
  | d <- parseDay3 ???{- begining of s -}
  , m <- parseMonth3 ???{- end of s -}]
```



Let the parsers decide

```
parseDate4 :: String -> [(Date,String)]
parseDate4 input = [(Date d m,tail')
  | (d,tail ) <- parseDay4 input
  , (m,tail') <- parseMonth4 tail]
```



Let the parsers decide

```
parseDate4 :: String -> [(Date,String)]
parseDate4 input = [(Date d m,tail')
  | (d,tail ) <- parseDay4 input
  , (m,tail') <- parseMonth4 tail]
```



Tidy up

```
parseDate4 :: String -> [(Date,String)]  
parseMonth4 :: String -> [(Month,String)]  
parseDay4 :: String -> [(Int,String)]
```

Repetitive types



Tidy up

```
parseDate5 :: Parser Date  
parseMonth5 :: Parser Month  
parseDay5 :: Parser Day  
  
type Parser a = String -> [(a,String)]
```



Tidy up

```
parseDate5 :: Parser Date
parseDate5 input = [(Date d m,tail)
  | (d,tail ) <- parseDay5 input
  , (m,tail') <- parseMonth5 tail]
```

Repetitive definitions



Tidy up

```
parseDate5 :: Parser Date
parseDate5 input = [(Date d m,tail)
  | (d,tail ) <- parseDay5 input
  , (m,tail') <- parseMonth5 tail]
```

Repetitive definitions



Tidy up

```
parseDate5 :: Parser Date
parseDate5 input = [(Date d m,tail')
  | (d,tail ) <- parseDay5 input
  , (m,tail') <- parseMonth5 tail]
```

Repetitive definitions



Tidy up

```
parseDate5 :: Parser Date
parseDate5 input = [(Date d m,tail')
  | (d,tail ) <- parseDay5 input
  , (m,tail') <- parseMonth5 tail]
```

Repetitive definitions

Goal:

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




Working out types

parseDate6 = Date <\$> parseDay <*> parseMonth



Working out types

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



Parser Date



Working out types

Int ->
Month ->
Date

parseDate6 = $\overbrace{\text{Date} \langle \$ \rangle \text{parseDay} \langle * \rangle \text{parseMonth}}^{\text{Parser Date}}$



Working out types

```
Int -> Parser Int
Month ->
Date

parseDate6 = Date <$> parseDay <*> parseMonth
              └──────────────────────────────────┘
                    Parser Date
```



Working out types

```
Int -> Parser Int
Month -> Parser Month
Date
```

parseDate6 = Date <\$> parseDay <*> parseMonth

Parser Date



Working out types

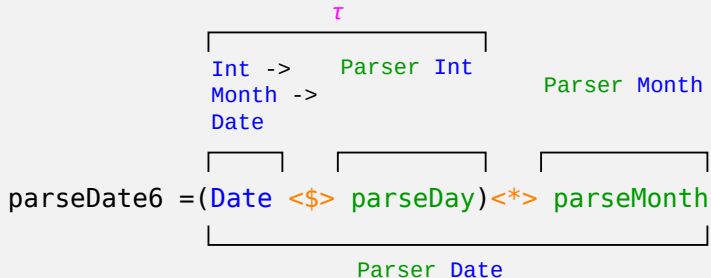
```
Int -> Parser Int
Month -> Parser Month
Date
```

parseDate6 = (Date <\$> parseDay) <*> parseMonth

Parser Date



Working out types



Working out types

τ

Int -> Parser Int
Month -> Parser Month
Date

parseDate6 = (Date <\$> parseDay) <*> parseMonth

Parser Date

<\$> :: (Int -> Month -> Date) -> Parser Int -> τ



Working out types

τ

```
Int -> Parser Int
Month -> Parser Month
Date
```

parseDate6 = (Date <\$> parseDay) <*> parseMonth

Parser Date

<\$> :: (Int -> Month -> Date) -> Parser Int -> τ
<*> :: τ -> Parser Month -> Parser Date



Working out types

```
<$> :: (Int -> Month -> Date) -> Parser Int ->  $\tau$   
<*> ::  $\tau$  -> Parser Month -> Parser Date
```



Working out types

$\langle \$ \rangle :: (\text{Int} \rightarrow \text{Month} \rightarrow \text{Date}) \rightarrow \text{Parser Int} \rightarrow \tau$

$\langle * \rangle :: \tau \rightarrow \text{Parser Month} \rightarrow \text{Parser Date}$



Working out types

```
<$> :: (Int -> Month -> Date)  
      -> Parser Int  
      ->  $\tau$ 
```

```
<*> ::  $\tau$   
      -> Parser Month  
      -> Parser Date
```



$\tau = \text{Parser (Month} \rightarrow \text{Date)}$

```
<$> :: (Int -> (Month -> Date))  
      -> Parser Int  
      -> Parser (Month -> Date)  
  
<*> :: Parser (Month -> Date)  
      -> Parser Month  
      -> Parser Date
```



$\tau = \text{Parser (Month} \rightarrow \text{Date)}$

$\langle \$ \rangle :: (\text{Int} \rightarrow (\text{Month} \rightarrow \text{Date}))$
 $\rightarrow \text{Parser Int}$
 $\rightarrow \text{Parser (Month} \rightarrow \text{Date)}$

$\langle * \rangle :: \text{Parser (Month} \rightarrow \text{Date)}$
 $\rightarrow \text{Parser Month}$
 $\rightarrow \text{Parser Date}$

$\$:: (\text{Month} \rightarrow \text{Date})$
 $\rightarrow \text{Month}$
 $\rightarrow \text{Date}$



Generalising

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



Generalising

$\langle \$ \rangle :: (a \rightarrow b)$

\rightarrow Parser a

\rightarrow Parser b

$\langle * \rangle :: \text{Parser } (a \rightarrow b)$

\rightarrow Parser a

\rightarrow Parser b

$\$:: (a \rightarrow b)$

\rightarrow a

\rightarrow b



Generalising

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$
 $\langle * \rangle :: \text{Parser } (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$
 $\$:: (a \rightarrow b) \rightarrow a \quad b$



Progress on parseDate

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

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

`parseDate6 :: Parser Date`

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



Progress on parseDate

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

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

`parseDate6 :: Parser Date`

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



Implementing $\langle \$ \rangle$

$\langle \$ \rangle ::=$

$(a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(\langle \$ \rangle) = ???$



Implementing $\langle \$ \rangle$

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(f \langle \$ \rangle \text{ parse}) = ???$



Implementing $\langle \$ \rangle$

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(f \langle \$ \rangle \text{ parse}) \text{ input} = ???$



Implementing $\langle \$ \rangle$

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(f \langle \$ \rangle \text{ parse}) \text{ input} = [\text{???} \mid \text{???}]$



Implementing $\langle \$ \rangle$

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(f \langle \$ \rangle \text{ parse}) \text{ input} = [(???, ???) \mid ???]$



Implementing $\langle \$ \rangle$

$\langle \$ \rangle :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

```
(f  $\langle \$ \rangle$  parse) input = [ (???, ???)
  | ??? <- parse input]
```



Implementing <\$>

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

```
(f <$> parse) input = [ (???,???)  
  | (x, tail) <- parse input]
```



Implementing <\$>

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

```
(f <$> parse) input = [ (f x,???)  
  | (x, tail) <- parse input]
```



Implementing $\langle \$ \rangle$

```
 $\langle \$ \rangle$  ::      (a -> b) -> Parser a -> Parser b
```

```
(f  $\langle \$ \rangle$  parse) input = [ (f x, tail)  
  | (x, tail) <- parse input]
```



Implementing <\$>

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

```
(f <$> parse) input = [ (f x, tail)  
  | (x, tail) <- parse input]
```

Examples



Implementing <\$>

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

```
(f <$> parse) input = [ (f x, tail)  
  | (x, tail) <- parse input]
```

Examples

```
((1+) <$> parseNat) "100" == [(101, "")]
```



Implementing <\$>

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

```
(f <$> parse) input = [ (f x, tail)  
  | (x, tail) <- parse input]
```

Examples

```
((1+) <$> parseNat) "100" == [(101, "")]  
(map toUpper <$> parseString "hello") "hello world"  
  == [("HELLO", " world")]
```



Implementing <\$>

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

```
(f <$> parse) input = [ (f x, tail)  
  | (x, tail) <- parse input]
```

Examples

```
((1+) <$> parseNat) "100" == [(101,"")]
```

```
(map toUpper <$> parseString "hello") "hello world"  
  == [("HELLO", " world")]
```

```
(length <$> parseString "hello") "hello world"  
  == [(5, " world")]
```



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
```



Implementing $\langle * \rangle$

$\langle * \rangle :: \text{Parser } (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(\langle * \rangle) = ???$



Implementing $\langle * \rangle$

$\langle * \rangle :: \text{Parser } (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$\text{pf } \langle * \rangle \text{ px} = ???$



Implementing $\langle * \rangle$

$\langle * \rangle :: \text{Parser } (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

$(\text{pf } \langle * \rangle \text{ px}) \text{ input} = ???$



Implementing <*>

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

(pf <*> px) input = [??? | ???]



Implementing <*>

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

(pf <*> px) input = [(???,???) | ???]



Implementing `<*>`

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

```
(pf <*> px) input = [ (???,???)
```

```
  | ??? <- pf ???
```

```
  , ??? <- px ???]
```



Implementing <*>

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

```
(pf <*> px) input = [ (???,???)  
  | (f, tail1) <- pf ???  
  , (x, tail2) <- px ???]
```



Implementing <*>

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

```
(pf <*> px) input = [ (f x,???)  
  | (f, tail1) <- pf ???  
  , (x, tail2) <- px ???]
```



Implementing <*>

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

```
(pf <*> px) input = [ (f x,???)  
  | (f, tail1) <- pf input  
  , (x, tail2) <- px tail1]
```



Implementing <*>

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

```
(pf <*> px) input = [ (f x, tail2)  
  | (f, tail1) <- pf input  
  , (x, tail2) <- px tail1]
```



Implementing <*>

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

```
(pf <*> px) input = [ (f x, tail2)  
  | (f, tail1) <- pf input  
  , (x, tail2) <- px tail1]
```

Example



Implementing <*>

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

```
(pf <*> px) input = [ (f x, tail2)  
  | (f, tail1) <- pf input  
  , (x, tail2) <- px tail1]
```

Example

```
(,) <$> parseNat <*> parseString " green bottles" $  
"42 green bottles hanging on the wall"  
== [((42," green bottles")," hanging on the wall")]
```



Progress on parseDate

```
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
```



Progress on parseDate

```
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
```



Progress on parseDate

```
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
```



parseDay

parseDay :: Parser Int



parseDay

```
parseDay :: Parser Int
```

```
parseDay "15" ✓
```

```
parseDay "2" ✓
```



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" ✓



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`





Why not “31 Dec” , “31 Feb”  ?

▶ “29 Feb” ?



Why not “31 Dec” , “31 Feb”  ?


▶ “29 Feb” 

▶  Dates are hard



Why not “31 Dec” , “31 Feb” ?

▶ “29 Feb” ?


▶  Dates are hard

▶ codeblog.jonskeet.uk/2015/05/05/



Why not “31 Dec” , “31 Feb” ?

▶ “29 Feb” ?

▶  Dates are hard

▶ codeblog.jonskeet.uk/2015/05/05/

▶ xkcd.com/1179/



Why not “31 Dec” , “31 Feb” ?

- ▶ “29 Feb” 
- ▶  Dates are hard
 - ▶ codeblog.jonskeet.uk/2015/05/05/
 - ▶ xkcd.com/1179/
- ▶ Instead parse **then** filter



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
```



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 = ???
```

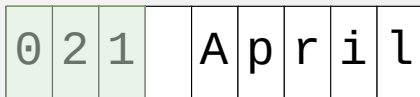
0	2	1		A	p	r	i	l
---	---	---	--	---	---	---	---	---



parsePositiveInt implementation

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

```
parsePositiveInt input = ???
```



parsePositiveInt implementation

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

```
parsePositiveInt input = ???
```

parsePositiveInt

("0 2 1 A p r i l")



parsePositiveInt implementation

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

```
parsePositiveInt input = ???
```

parsePositiveInt

("0 2 1 April")

== [(21, "April")]



parsePositiveInt implementation

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

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

parsePositiveInt

("0 2 1 April")

== [(21, "April")]

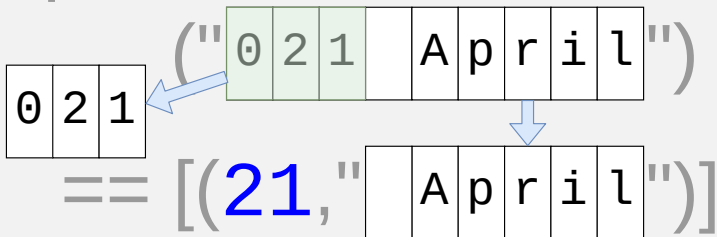


parsePositiveInt implementation

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

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

parsePositiveInt

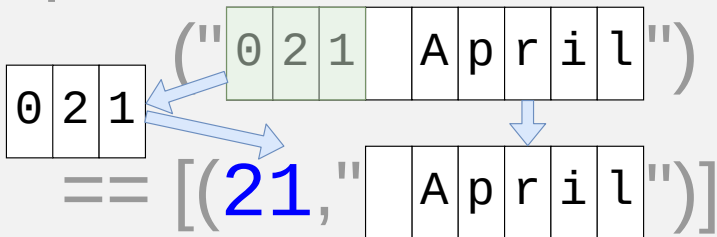


parsePositiveInt implementation

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

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

parsePositiveInt

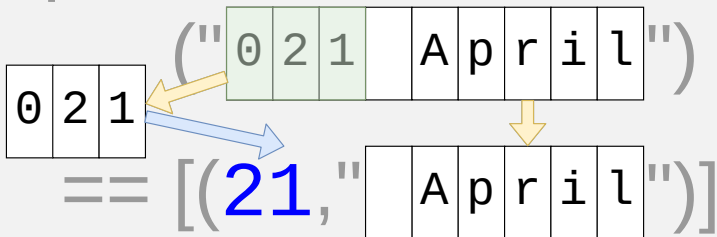


parsePositiveInt implementation

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

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

parsePositiveInt



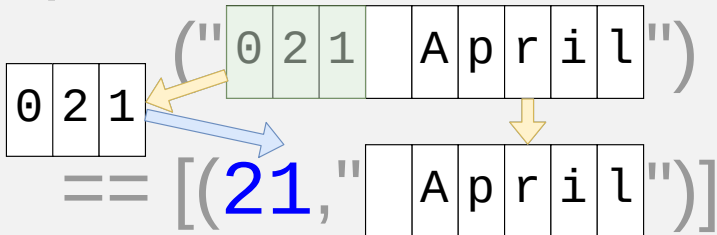
parsePositiveInt implementation

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

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

```
  where (numStr, tail) = ??? input
```

parsePositiveInt

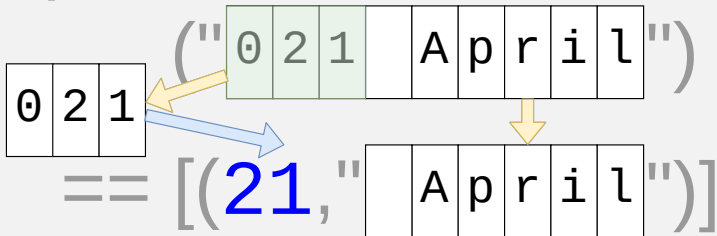


parsePositiveInt implementation

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

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

parsePositiveInt



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)
```



Search plugin | Manual | haskell.org

Hoogle

Packages

- is:exact
- basement
- utf8-string
- ghc
- ghc-lib-parser

Results:

span :: (Char -> Bool) -> String -> (String, String)
basement Basement.String
Ⓢ Apply a predicate to the string to return the longest prefix that satisfy the predicate and the remaining

spanEnd :: (Char -> Bool) -> String -> (String, String)
basement Basement.String
Ⓢ Apply a predicate to the string to return the longest suffix that satisfy the predicate and the remaining



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)
```



Hoogle

Packages

- is:exact
- basement
- utf8-string
- ghc
- ghc-lib-parser

span :: (Char -> Bool) -> String -> (String, String)
basement Basement.String
Ⓢ Apply a predicate to the string to return the longest prefix that satisfy the predicate and the remaining

spanEnd :: (Char -> Bool) -> String -> (String, String)
basement Basement.String
Ⓢ Apply a predicate to the string to return the longest suffix that satisfy the predicate and the remaining



Aside: how to guess

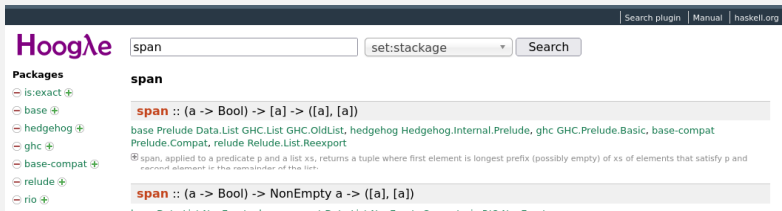
```
parsePositiveInt :: String -> [(Int,String)]
```

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

```
  where (numStr, tail) = Basement.String.span ???_2 inp
```

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

```
???)_2 :: (Char -> Bool)
```



The screenshot shows the Hoogle search engine interface. The search bar contains the word "span". The search results are displayed under the heading "span". The first result shows the type signature: `span :: (a -> Bool) -> [a] -> ([a], [a])`. Below this, it lists the modules where the function is defined: `base Prelude Data.List GHC.List GHC.OldList, hedgehog Hedgehog.Internal.Prelude, ghc GHC.Prelude.Basic, base-compat Prelude.Compat, relude Relude.List.Reexport`. A description follows: "span, applied to a predicate p and a list xs, returns a tuple where first element is longest prefix (possibly empty) of xs of elements that satisfy p and second element is the remainder of the list." The second result shows the type signature: `span :: (a -> Bool) -> NonEmpty a -> ([a], [a])`. Below this, it lists the modules: `base Data.List NonEmpty, base-compat Data.List NonEmpty Compat, rio RIO NonEmpty`.



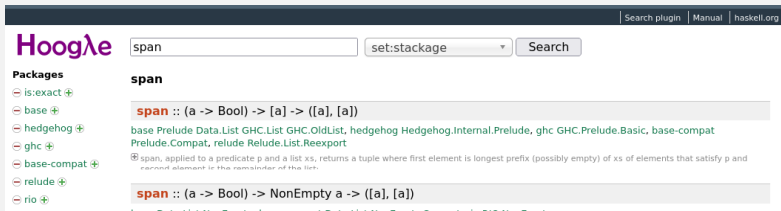
Aside: how to guess

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

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

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

```
???2 :: (Char -> Bool)
```



The screenshot shows the Hoogle search interface. The search bar contains the text "span". The search results are displayed under the heading "span". The first result is the type signature: `span :: (a -> Bool) -> [a] -> ([a], [a])`. Below this, the source modules are listed: `base Prelude Data.List GHC.List GHC.OldList, hedgehog Hedgehog.Internal.Prelude, ghc GHC.Prelude.Basic, base-compat Prelude.Compat, relude Relude.List.Reexport`. A description follows: `span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list. The second result is another type signature: `span :: (a -> Bool) -> NonEmpty a -> ([a], [a])`. Below this, the source modules are listed: `base Data.List NonEmpty, base-compat Data.List NonEmpty Compat, rio RIO NonEmpty`.



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)
```



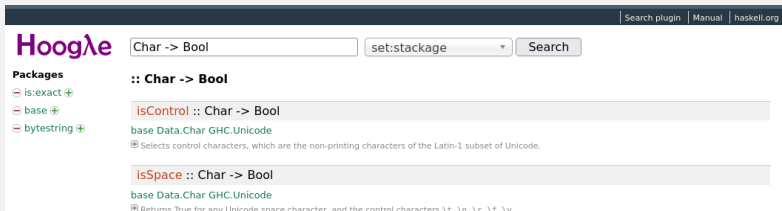
Aside: how to guess

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

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

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

```
???2 :: (Char -> Bool)
```



The screenshot shows the Hoogle search engine interface. At the top, there is a search bar containing the text "Char -> Bool", a dropdown menu set to "set:stackage", and a "Search" button. Below the search bar, the "Packages" section is visible, listing several packages with expand/collapse icons. The search results are displayed as follows:

- isControl :: Char -> Bool**
base Data.Char GHC.Unicode
Selects control characters, which are the non-printing characters of the Latin-1 subset of Unicode.
- isSpace :: Char -> Bool**
base Data.Char GHC.Unicode
Returns True for any Unicode space character, and the control characters \t, \n, \r, \f, \v.



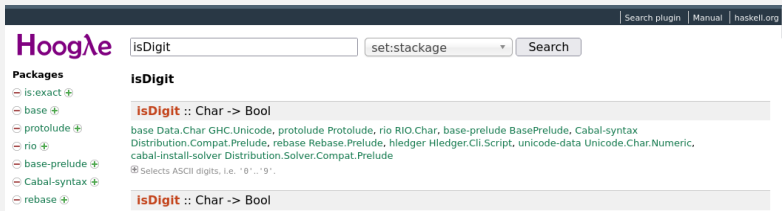
Aside: how to guess

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

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

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

```
???2 :: (Char -> Bool)
```



The screenshot shows the Hoogle search engine interface. The search bar contains 'isDigit' and the dropdown menu is set to 'stackage'. The search results are displayed under the heading 'isDigit'. The first result is 'isDigit :: Char -> Bool' with a description: 'base Data.Char GHC.Unicode, protolude Protolude, rio RIO.Char, base-prelude BasePrelude, Cabal-syntax Distribution.Compat.Prelude, rebase Rebase.Prelude, hledger Hledger.Cli.Script, unicode-data Unicode.Char.Numeric, cabal-install-solver Distribution.Solver.Compat.Prelude'. A note below the description states: 'Selects ASCII digits, i.e. '0'..'9''. The second result is also 'isDigit :: Char -> Bool'.



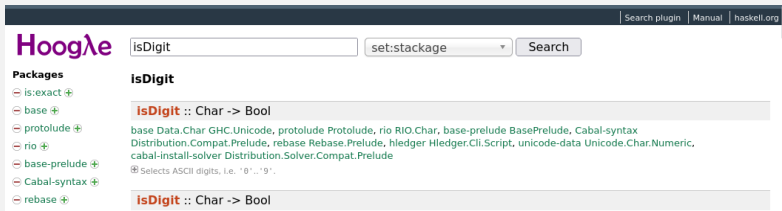
Aside: how to guess

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

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

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

```
isDigit :: (Char -> Bool)
```



The screenshot shows the Hoogle search engine interface. At the top right, there are links for "Search plugin", "Manual", and "haskell.org". The search bar contains the text "isDigit". To the right of the search bar is a dropdown menu set to "set:stackage" and a "Search" button. On the left side, under the "Packages" section, there is a list of packages with expand/collapse icons: is:exact, base, protolude, rio, base-prelude, Cabal-syntax, and rebase. The main search results area shows two entries for "isDigit :: Char -> Bool". The first entry lists several packages: base, Data.Char, GHC.Unicode, protolude, rio, RIO.Char, base-prelude, BasePrelude, Cabal-syntax, Distribution.Compat.Prelude, rebase, Rebase.Prelude, hledger, Hledger.Cli.Script, unicode-data, Unicode.Char.Numeric, and cabal-install-solver. Below this list is a note: "⊗ Selects ASCII digits, i.e. '0'..'9'". The second entry is a duplicate of the first.



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)
```

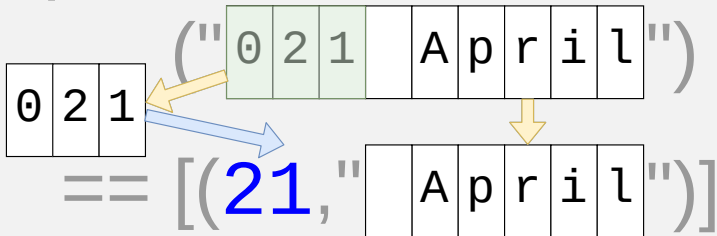


parsePositiveInt implementation

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

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

parsePositiveInt

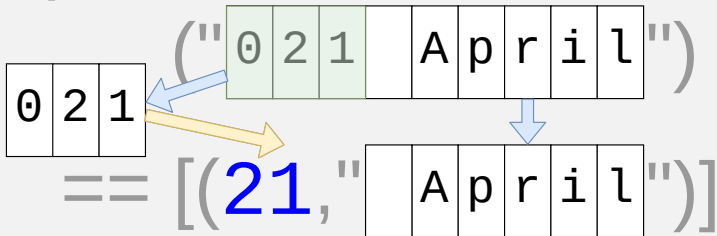


parsePositiveInt implementation

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

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

parsePositiveInt



parsePositiveInt implementation

```
parsePositiveInt :: String -> [(Int,String)]  
parsePositiveInt input = [(???, tail)]  
  where (numStr, tail) = span isDigit input  
  
numStr :: String  
???: Int
```



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
```



parsePositiveInt testing

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



parsePositiveInt testing

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

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



parsePositiveInt testing

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

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

```
parsePositiveInt "1 April" == [(1, " April")] ✓
```



parsePositiveInt testing

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

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

```
parsePositiveInt "1 April" == [(1, " April")] ✓
```

```
parsePositiveInt "007" == [(7, "")] ✓
```



parsePositiveInt testing

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

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

```
parsePositiveInt "1 April" == [(1, " April")] ✓
```

```
parsePositiveInt "007" == [(7, "")] ✓
```

```
parsePositiveInt "0" == [(0, "")] ✗
```



parsePositiveInt testing

```
parseNat :: Parser Int
```

```
parseNat input = [(read numStr, tail)]
```

```
  where (numStr, tail) = span isDigit input
```

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

```
parsePositiveInt "1 April" == [(1, " April")] ✓
```

```
parsePositiveInt "007" == [(7, "")] ✓
```

```
parsePositiveInt "0" == [(0, "")] ✗
```



parsePositiveInt testing

```
parseNat :: Parser Int
```

```
parseNat input = [(read numStr, tail)]
```

```
  where (numStr, tail) = span isDigit input
```

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

```
parsePositiveInt "1 April" == [(1, " April")] ✓
```

```
parsePositiveInt "007" == [(7, "")] ✓
```

```
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

```
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

```
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

```
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,"")] ✓
```



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 not a number"
        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) = 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 number"  
          in [(read numStr, 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 numStr = "  
          tail = "this is not a number"  
        in [(read numStr, 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]
```





```
== [ (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

```
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")] (✓)
```



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")] (✓)
```

Exercise

- ▶ Reimplement `parsePositiveInt` by combining simpler parsers
- ▶ **Hint:** Experiment with `greedy1`



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" ✓`



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")] ✓
```



Implementing parseMonth

```
parseMonth :: Parser Month
```

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

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

But ugly!



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

```
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

```
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")] ✓
```



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")] ✓
```



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

```
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 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" == [] ✗
```



Testing parseMonth

```
parseMonth :: Parser Month
```

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

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

```
parseMonth "jul" == [] ✗
```



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

```
parseMonth :: Parser Month
```

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

```
parsePrefixOf :: String -> Parser String
```

```
parsePrefixOf "july" "july" ✓
```



Designing parsePrefixOf

```
parseMonth :: Parser Month
```

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

```
parsePrefixOf :: String -> Parser String
```

```
parsePrefixOf "july" "july" ✓
```

```
parsePrefixOf "july" "jul" ✓
```



Designing parsePrefixOf

```
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 String
```

```
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 String
```

```
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 String
```

```
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 String
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 String
```

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

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

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

```
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 (/= "") (parseLongest
```



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" "" == [("","")] ✓
```



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")] ✓
```



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")] ✓
```



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")] ✓
```

Exercise

- ▶ Implement parseLongestPrefixOf by combining simpler

parsers

- ▶ Hint: experiment with <<|>



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" == [] ✅
```



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 $\langle | \rangle$

$\langle | \rangle$ `:: 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" 
```



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 -> 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"  
== [] ✓
```



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, "")] ✓
```



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) output
```

```
-- 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) output
```

```
-- 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) output
```

```
-- 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) output
```

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

```
oneGroup = take 1 grouped
```

```
-- e.g. [[(June, "")]]
```

```
concatened = concat firstGroup
```

[Faculty of Science
Information and Computing
Sciences]



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) output
```

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

```
oneGroup = take 1 grouped
```

```
-- e.g. [[(June, "")]]
```

```
concatened = concat firstGroup
```

[Faculty of Science
Information and Computing
Sciences]



```
-- e.g. [(June, "")]
```

Implementing longest

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

```
longest parser input = concatenated
```

```
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) output
```

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

```
oneGroup = take 1 grouped
```

```
-- e.g. [[(June, "")]]
```

```
concatened = concat firstGroup
```

[Faculty of Science
Information and Computing
Sciences]



```
-- e.g. [[(June, "")]]
```

Tidying up longest

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

```
longest parser input = concatenated
```

```
  where outputs = parser input
```

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

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

```
  oneGroup = take 1 grouped
```

```
concatened = concat firstGroup
```

[Faculty of Science
Information and Computing
Sciences]



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) sorted
```

```
        firstGroup = take 1 grouped
```

```
        concatenated = 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) sorted
```

```
        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) sorted
```



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

```
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

```
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

```
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

```
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")]
```

[Faculty of Science
Information and Computing
Sciences]



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 <$> parseLongestNEPrefixOf "noven  
<|> const December <$> 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"
```

```
== [] X
```



Testing parseDate

```
parseDate6 :: Parser Date
```

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

```
parseDate6 "31 oct"
```

```
== [] ❌
```

```
parseDate6 "31oct"
```

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



Testing parseDate

```
parseDate6 :: Parser Date
```

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

```
parseDate6 "31 oct"
```

```
== [] ❌
```

```
parseDate6 "31oct"
```

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



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

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



Handling Whitespace another way

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

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



Handling Whitespace another way

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

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

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



Handling Whitespace another way

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



Handling Whitespace another way

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

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

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



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, "") ] ✓
```



Summary

```
parseDay :: Parser Int
```

```
parseMonth :: Parser Month
```

```
<$ :: Parser 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

