



Functional Programming

Lecture 9

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

Topics

abstract data types, algebraic data types, binary search trees, **combinator parsing**, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, **infinite data structures**, input and output, lambda-calculus, **lazy evaluation**, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, **type inference**, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, **combinator parsing**, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, **infinite data structures**, input and output, lambda-calculus, **lazy evaluation**, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Parsing
- Combinator Parsing
- Parsing XML Data

Parsing

What is Parsing?

- decomposition of **sequence of symbols**

What is Parsing?

- decomposition of sequence of symbols
- according to **grammar**

What is Parsing?

- decomposition of sequence of symbols
- according to grammar
- resulting in **structured data**

What is Parsing?

- decomposition of sequence of symbols
- according to grammar
- resulting in structured data

Example

(x o n e + x t w o)

What is Parsing?

- decomposition of sequence of symbols
- according to grammar
- resulting in structured data

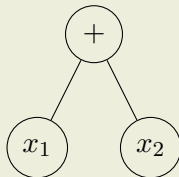
Example

(xone + xtwo)

What is Parsing?

- decomposition of sequence of symbols
- according to grammar
- resulting in structured data

Example



Potential Sequences of Symbols

- text in natural language

Potential Sequences of Symbols

- text in natural language
- source code of a computer program

Potential Sequences of Symbols

- text in natural language
- source code of a computer program
- a website

Potential Sequences of Symbols

- text in natural language
- source code of a computer program
- a website
- a sequence of genes

Potential Sequences of Symbols

- text in natural language
- source code of a computer program
- a website
- a sequence of genes
- ...

Potential Sequences of Symbols

- text in natural language
- source code of a computer program
- a website
- a sequence of genes
- ...

In the Following

- sequence of symbols: a list of so called **tokens** (type `[t]`)
- grammar: Backus–Naur Form (BNF)
- structured data: some user defined data type

BNF of XML Data (Simplified)

$$\langle xml \rangle ::= \langle \langle name \rangle \rangle \langle xml \rangle^* \langle / \langle name \rangle \rangle$$
$$| \langle text \rangle$$
$$\langle name \rangle ::= (\langle letter \rangle | _)(\langle letter \rangle | _ | \langle digit \rangle)^*$$
$$\langle letter \rangle ::= a | \dots | z | A | \dots | Z$$
$$\langle digit \rangle ::= 0 | \dots | 9$$
$$\langle text \rangle ::= (\text{"every symbol except for <"})^+$$

BNF of XML Data (Simplified)

$$\langle xml \rangle ::= \langle \langle name \rangle \rangle \langle xml \rangle^* \langle / \langle name \rangle \rangle$$
$$| \langle text \rangle$$
$$\langle name \rangle ::= (\langle letter \rangle | _)(\langle letter \rangle | _ | \langle digit \rangle)^*$$
$$\langle letter \rangle ::= a | \dots | z | A | \dots | Z$$
$$\langle digit \rangle ::= 0 | \dots | 9$$
$$\langle text \rangle ::= (\text{"every symbol except for <"})^+$$

Example from W3Schools

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Parsers – First Attempt

- functions of type $[t] \rightarrow (a, [t])$

Parsers – First Attempt

- functions of type $[t] \rightarrow (a, [t])$
- read tokens from given list and produce result (of type a) together with list of remaining tokens

Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`

Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`
- but what about errors? (like for `digit "abc"`)

Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`
- but what about errors? (like for `digit "abc"`)

Type of Parsers

- use `newtype` to distinguish from similar function types

```
newtype Parser t a =  
  Parser { run :: [t] -> Maybe (a, [t]) }
```


Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`
- but what about errors? (like for `digit "abc"`)

Type of Parsers

- use `newtype` to distinguish from similar function types
`newtype Parser t a =`
 `Parser { run :: [t] -> Maybe (a, [t]) }`
- parser works on list of tokens of arbitrary type `t`

Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`
- but what about errors? (like for `digit "abc"`)

Type of Parsers

- use `newtype` to distinguish from similar function types
`newtype Parser t a =
 Parser { run :: [t] -> Maybe (a, [t]) }`
- parser works on list of tokens of arbitrary type `t`
- successful parse yields `Just (x, ts)` with result `x` and remaining tokens `ts`

Parsers – First Attempt

- functions of type `[t] -> (a, [t])`
- read tokens from given list and produce result (of type `a`) together with list of remaining tokens
- for example, `digit "12"` might result in `('1', "2")`
- but what about errors? (like for `digit "abc"`)

Type of Parsers

- use `newtype` to distinguish from similar function types
`newtype Parser t a =
 Parser { run :: [t] -> Maybe (a, [t]) }`
- parser works on list of tokens of arbitrary type `t`
- successful parse yields `Just (x, ts)` with result `x` and remaining tokens `ts`
- error yields `Nothing` (no exact error message)

Note

- traditionally parsing split into two phases

Note

- traditionally parsing split into two phases

Lexing

- divide original input (list of **Chars**) into other type of tokens
- white space characters and comments often dropped at this stage

Note

- traditionally parsing split into two phases

Lexing

- divide original input (list of **Chars**) into other type of tokens
- white space characters and comments often dropped at this stage

Parsing

- actual parser works on list of tokens provided by lexer
- typically produces structured data

Note

- traditionally parsing split into two phases
- combinator parsers applicable to both stages

Lexing

- divide original input (list of **Chars**) into other type of tokens
- white space characters and comments often dropped at this stage

Parsing

- actual parser works on list of tokens provided by lexer
- typically produces structured data

Tokens for XML Data

```
data Token = StartTag String -- <a>, <example>, ...
           | EndTag String   -- </a>, </example>, ...
           | Comment String  -- <!-- arbitrary text -->
           | Text String
deriving Show
```


Tokens for XML Data

```
data Token = StartTag String -- <a>, <example>, ...
           | EndTag String   -- </a>, </example>, ...
           | Comment String  -- <!-- arbitrary text -->
           | Text String
deriving Show
```

Data Type for XML Data

```
type Tag = String
data Xml = Xml Tag [Xml]
         | Txt String
deriving Show
```

Tokens for XML Data

```
data Token = StartTag String -- <a>, <example>, ...
           | EndTag String   -- </a>, </example>, ...
           | Comment String  -- <!-- arbitrary text -->
           | Text String

deriving Show
```

Data Type for XML Data

```
type Tag = String
data Xml = Xml Tag [Xml]
         | Txt String

deriving Show
```

Example

- XML document:

```
<ul>
  <li>some thing</li>
  <li>another thing</li>
</ul>
```
- representation with `Xml` data type:

```
Xml "ul" [
  Xml "li" [Txt "some thing"],
  Xml "li" [Txt "another thing"]]
```

Combinator Parsing

Running Parsers on Input

- apply parser to list of tokens

```
parse :: Parser t a -> [t] -> Maybe a
```

```
parse p ts = case run p ts of
```

```
  Just (x, _) -> Just x
```

```
  Nothing     -> Nothing
```

Running Parsers on Input

- apply parser to list of tokens

```
parse :: Parser t a -> [t] -> Maybe a
```

```
parse p ts = case run p ts of
```

```
  Just (x, _) -> Just x
```

```
  Nothing      -> Nothing
```

- for testing purposes

```
test :: Parser t a -> [t] -> a
```

```
test p ts = case parse p ts of
```

```
  Just x -> x
```

```
  Nothing -> error "Parse.test: parse error"
```

Primitive Parsers

- **primitive parsers** need to know about implementation

Primitive Parsers

- primitive parsers need to know about implementation
- turn arbitrary value into parser (“lift type `a` into `Parser t a`”)
`lift :: a -> Parser t a`
`lift x = Parser $ \ts -> Just (x, ts)`

Primitive Parsers

- primitive parsers need to know about implementation
- turn arbitrary value into parser (“lift type `a` into `Parser t a`”)

```
lift :: a -> Parser t a
```

```
lift x = Parser $ \ts -> Just (x, ts)
```

- accept single token specified by function

```
token :: (t -> Maybe a) -> Parser t a
```

```
token f = Parser $ \ts -> case ts of
```

```
  [] -> Nothing
```

```
  x:xs -> case f x of
```

```
    Just y -> Just (y, xs)
```

```
    Nothing -> Nothing
```


Primitive Parsers

- primitive parsers need to know about implementation
- turn arbitrary value into parser (“lift type `a` into `Parser t a`”)

```
lift :: a -> Parser t a
```

```
lift x = Parser $ \ts -> Just (x, ts)
```

- accept single token specified by function

```
token :: (t -> Maybe a) -> Parser t a
```

```
token f = Parser $ \ts -> case ts of
```

```
  [] -> Nothing
```

```
  x:xs -> case f x of
```

```
    Just y -> Just (y, xs)
```

```
    Nothing -> Nothing
```

- only accept end of input

```
eoi :: Parser t ()
```

```
eoi = Parser $ \ts -> case ts of
```

```
  [] -> Just ((), [])
```

```
  x:xs -> Nothing
```

Derived Parsers (implementation agnostic)

- parsing single tokens

```
sat :: (t -> Bool) -> Parser t t
```

```
sat p = token $ \t -> if p t then Just t else Nothing
```

```
anyToken :: Parser t t
```

```
anyToken = sat (const True)
```

Derived Parsers (implementation agnostic)

- parsing single tokens

```
sat :: (t -> Bool) -> Parser t t
```

```
sat p = token $ \t -> if p t then Just t else Nothing
```

```
anyToken :: Parser t t
```

```
anyToken = sat (const True)
```

- parse specific character

```
char :: Char -> Parser Char Char
```

```
char c = sat (== c)
```

Derived Parsers (implementation agnostic)

- parsing single tokens

```
sat :: (t -> Bool) -> Parser t t
```

```
sat p = token $ \t -> if p t then Just t else Nothing
```

```
anyToken :: Parser t t
```

```
anyToken = sat (const True)
```

- parse specific character

```
char :: Char -> Parser Char Char
```

```
char c = sat (== c)
```

- accepting/rejecting with respect to list of tokens

```
oneof cs = sat (`elem` cs)
```

```
noneof cs = sat (`notElem` cs)
```

Derived Parsers (implementation agnostic)

- parsing single tokens

```
sat :: (t -> Bool) -> Parser t t
```

```
sat p = token $ \t -> if p t then Just t else Nothing
```

```
anyToken :: Parser t t
```

```
anyToken = sat (const True)
```

- parse specific character

```
char :: Char -> Parser Char Char
```

```
char c = sat (== c)
```

- accepting/rejecting with respect to list of tokens

```
oneof cs = sat (`elem` cs)
```

```
noneof cs = sat (`notElem` cs)
```

- parsing letters and digits

```
letter = oneof (['a'..'z']++['A'..'Z'])
```

```
digit = oneof ['0'..'9']
```

Derived Parsers (implementation agnostic)

- parsing single tokens

```
sat :: (t -> Bool) -> Parser t t
```

```
sat p = token $ \t -> if p t then Just t else Nothing
```

```
anyToken :: Parser t t
```

```
anyToken = sat (const True)
```

- parse specific character

```
char :: Char -> Parser Char Char
```

```
char c = sat (== c)
```

- accepting/rejecting with respect to list of tokens

```
oneof cs = sat (`elem` cs)
```

```
noneof cs = sat (`notElem` cs)
```

- parsing letters and digits

```
letter = oneof (['a'..'z']++['A'..'Z'])
```

```
digit = oneof ['0'..'9']
```

- parsing single white space characters

```
space = oneof " \n\r\t"
```

Primitive Parser Combinators – Choice

- (parser) combinator builds parser from one (or more) given parser(s)

Primitive Parser Combinators – Choice

- (parser) combinator builds parser from one (or more) given parser(s)
- definition of choice combinator

```
(<|>) :: Parser t a -> Parser t a -> Parser t a
p <|> q = Parser $ \ts ->
  case run p ts of
    Nothing -> run q ts
    r        -> r
```


Primitive Parser Combinators – Choice

- (parser) combinator builds parser from one (or more) given parser(s)
- definition of choice combinator

```
(<|>) :: Parser t a -> Parser t a -> Parser t a
p <|> q = Parser $ \ts ->
  case run p ts of
    Nothing -> run q ts
    r        -> r
```

- $p <|> q$ – parser that first tries p and on failure tries q

Primitive Parser Combinators – Choice

- (parser) combinator builds parser from one (or more) given parser(s)
- definition of choice combinator

```
(<|>) :: Parser t a -> Parser t a -> Parser t a
p <|> q = Parser $ \ts ->
  case run p ts of
    Nothing -> run q ts
    r        -> r
```

- $p <|> q$ – parser that first tries p and on failure tries q

Example

- $\langle p \rangle ::= a \mid b$
- $p = \text{char 'a'} <|> \text{char 'b'}$
- that is, $<|>$ corresponds to $|$ in BNF

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`
- `bind p f` – parser that first executes `p` and then feeds result into `f`

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`
- `bind p f` – parser that first executes `p` and then feeds result into `f`
- since `f` is function producing a parser, result of `bind p f` is parser

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`
- `bind p f` – parser that first executes `p` and then feeds result into `f`
- since `f` is function producing a parser, result of `bind p f` is parser

Example

- $\langle p \rangle ::= ab$

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`
- `bind p f` – parser that first executes `p` and then feeds result into `f`
- since `f` is function producing a parser, result of `bind p f` is parser

Example

- $\langle p \rangle ::= ab$
- `p = char 'a' `bind` _ -> char 'b'`

Primitive Parser Combinators – Sequencing

- definition

```
bind ::
```

```
  Parser t a -> (a -> Parser t b) -> Parser t b
```

```
bind p f = Parser $ \ts ->
```

```
  case run p ts of
```

```
    Just (x, ts') -> run (f x) ts'
```

```
    Nothing       -> Nothing
```

- `bind` takes parser with results of type `a`
- and function taking `a` and producing parser with results of type `b`
- `bind p f` – parser that first executes `p` and then feeds result into `f`
- since `f` is function producing a parser, result of `bind p f` is parser

Example

- $\langle p \rangle ::= ab$
- `p = char 'a' `bind` _ -> char 'b'`
- `char 'a' `bind` \x -> char 'b' `bind` \y -> lift [x,y]`

do-Notation for Parsers

- parsers are in some respects very similar to IO actions

do-Notation for Parsers

- parsers are in some respects very similar to IO actions
- instead of reading input and writing output, parsers read tokens and yield remaining tokens

do-Notation for Parsers

- parsers are in some respects very similar to IO actions
- instead of reading input and writing output, parsers read tokens and yield remaining tokens
- like IO actions, parsers can be run in sequence and arbitrary values can be turned (“lifted”) to parsers (using `lift`)

do-Notation for Parsers

- parsers are in some respects very similar to IO actions
- instead of reading input and writing output, parsers read tokens and yield remaining tokens
- like IO actions, parsers can be run in sequence and arbitrary values can be turned (“lifted”) to parsers (using `lift`)
- this pattern (sequencing and lifting) is so common that there is a dedicated type class

do-Notation for Parsers

- parsers are in some respects very similar to IO actions
- instead of reading input and writing output, parsers read tokens and yield remaining tokens
- like IO actions, parsers can be run in sequence and arbitrary values can be turned (“lifted”) to parsers (using `lift`)
- this pattern (sequencing and lifting) is so common that there is a dedicated type class

The Monad Class – Supporting do-Notation

- class functions
`return` :: `Monad m => a -> m a`
`(>>=)` :: `Monad m => m a -> (a -> m b) -> m b`
- `return` – lifts arbitrary value into monad
- `(>>=)` – (called “bind”) executes two monads one after the other, where second may depend on “output” of first

Monads and do-Notation

- do-notation is syntactic sugar for calls to ($\gg=$)
- translation uses following equalities (from top to bottom):

$$\mathbf{do} \{ \mathbf{let} \ x = e; M \} = \mathbf{let} \ x = e \ \mathbf{in} \ \mathbf{do} \{ M \}$$

$$\mathbf{do} \{ x \leftarrow m; M \} = m \gg= (\backslash x \rightarrow \mathbf{do} \{ M \})$$

$$\mathbf{do} \{ m; M \} = m \gg= (\backslash _ \rightarrow \mathbf{do} \{ M \})$$

$$\mathbf{do} \{ M \} = M$$

Monads and do-Notation

- do-notation is syntactic sugar for calls to ($\gg=$)
- translation uses following equalities (from top to bottom):

$$\mathbf{do} \{ \mathbf{let} \ x = e; M \} = \mathbf{let} \ x = e \ \mathbf{in} \ \mathbf{do} \{ M \}$$
$$\mathbf{do} \{ x \leftarrow m; M \} = m \gg= (\backslash x \rightarrow \mathbf{do} \{ M \})$$
$$\mathbf{do} \{ m; M \} = m \gg= (\backslash _ \rightarrow \mathbf{do} \{ M \})$$
$$\mathbf{do} \{ M \} = M$$

Example – IO

- do-block

```
do input <- readLn
  putStrLn ("input = " ++ input ++ "")
  let n = (read input :: Int)
  return n
```

- is transformed into

```
readLn >>= \input ->
  putStrLn ("input = " ++ input ++ "") >>= \_ ->
  let n = (read input :: Int)
  in return n
```

Parsers are Monads

```
instance Monad (Parser t) where
  return = lift
  (>>=) = bind
```

Parsers are Monads

```
instance Monad (Parser t) where
  return = lift
  (>>=) = bind
```

Okay ... but what does that mean?

- can use do-notation for parsers

Parsers are Monads

```
instance Monad (Parser t) where
  return = lift
  (>>=) = bind
```

Okay ... but what does that mean?

- can use do-notation for parsers
- allows us to apply parsers in sequence

Parsers are Monads

```
instance Monad (Parser t) where
  return = lift
  (>>=) = bind
```

Okay ... but what does that mean?

- can use do-notation for parsers
- allows us to apply parsers in sequence
- access result of parser using <-

Parsers are Monads

```
instance Monad (Parser t) where
  return = lift
  (>>=) = bind
```

Okay ... but what does that mean?

- can use do-notation for parsers
- allows us to apply parsers in sequence
- access result of parser using <-
- and turn arbitrary values into parser using `return`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$

Derived Combinators – Repetition

- **many** p applies p zero or more times
- result is list of results of p invocations
- greedy (as many applications of p as possible)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)
- `q = many1 (char 'a')`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application
- `manyTill p e`, similar to `many`, but stop at `e`

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p` = `many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)
- `q` = `many1 (char 'a')`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application
- `manyTill p e`, similar to `many`, but stop at `e`
- `string s` accepts specific string `s` (e.g., useful for parsing keywords)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)
- `q = many1 (char 'a')`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application
- `manyTill p e`, similar to `many`, but stop at `e`
- `string s` accepts specific string `s` (e.g., useful for parsing keywords)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)
- `q = many1 (char 'a')`
- arbitrary symbols until end-of-comment marker `-->`

Derived Combinators – Repetition

- `many p` applies `p` zero or more times
- result is list of results of `p` invocations
- greedy (as many applications of `p` as possible)
- `many1`, similar to `many`, but at least 1 application
- `manyTill p e`, similar to `many`, but stop at `e`
- `string s` accepts specific string `s` (e.g., useful for parsing keywords)

Examples

- $\langle p \rangle ::= a\langle p \rangle \mid \varepsilon$ (usually written a^*)
- `p = many (char 'a')`
- $\langle q \rangle ::= a\langle q \rangle \mid a$ (usually written a^+)
- `q = many1 (char 'a')`
- arbitrary symbols until end-of-comment marker `-->`
- `r = manyTill anyToken (string "-->")`

Derived Combinator

- apply a parser between two others

`between ::`

```
Parser t a -> Parser t b -> Parser t c  
    -> Parser t c
```

```
between l r p = do
```

```
  l
```

```
  x <- p
```

```
  r
```

```
  return x
```

Derived Combinator

- apply a parser between two others

`between ::`

```
Parser t a -> Parser t b -> Parser t c  
  -> Parser t c
```

`between l r p = do`

```
  l
```

```
  x <- p
```

```
  r
```

```
  return x
```

Example

- $\langle p \rangle ::= (a^*)$
- `p = between (char '(') (char ')') (many (char 'a'))`

Parsing XML Data

Recognizing Tokens – XML Tags

```
parseName :: Parser Char String
```

```
parseName = do
```

```
  x <- letter <|> char '_'
```

```
  xs <- many (letter <|> char '_' <|> digit)
```

```
  return $ x:xs
```

```
parseTag :: Parser Char Token
```

```
parseTag =
```

```
  between (char '<') (char '>' >> spaces) (sTag <|> eTag)
```

```
  where
```

```
    sTag = parseName >>= return . StartTag
```

```
    eTag = char '/' >> parseName >>= return . EndTag
```


Recognizing Tokens – XML Tags

```
parseName :: Parser Char String
```

```
parseName = do
```

```
  x <- letter <|> char '_'
```

```
  xs <- many (letter <|> char '_' <|> digit)
```

```
  return $ x:xs
```

```
parseTag :: Parser Char Token
```

```
parseTag =
```

```
  between (char '<') (char '>' >> spaces) (sTag <|> eTag)
```

```
  where
```

```
    sTag = parseName >>= return . StartTag
```

```
    eTag = char '/' >> parseName >>= return . EndTag
```

Note

- `p >> q` abbreviates `p >>= _ -> q`

Recognizing Tokens – XML Tags

```
parseName :: Parser Char String
```

```
parseName = do
```

```
  x <- letter <|> char '_'
```

```
  xs <- many (letter <|> char '_' <|> digit)
```

```
  return $ x:xs
```

```
parseTag :: Parser Char Token
```

```
parseTag =
```

```
  between (char '<') (char '>' >> spaces) (sTag <|> eTag)
```

```
  where
```

```
    sTag = parseName >>= return . StartTag
```

```
    eTag = char '/' >> parseName >>= return . EndTag
```

Note

- `p >> q` abbreviates `p >>= _ -> q`
- `spaces = many space >> return ()`

Recognizing Tokens – Comments and Text

```
parseComment :: Parser Char Token
parseComment = do
  string "<!--"
  cmt <- manyTill anyToken (string "-->")
  spaces
  return $ Comment cmt

parseText :: Parser Char Token
parseText = many1 (noneof "<") >>= return . Text
```

Lexing / Tokenization

```
lexer :: Parser Char [Token]
```

```
lexer = do
```

```
  spaces
```

```
  ts <- many (parseTag <|> parseComment <|> parseText)
```

```
  eoi
```

```
  return ts
```

```
tokenize :: [Char] -> Maybe [Token]
```

```
tokenize cs = case parse lexer cs of
```

```
  Nothing -> Nothing
```

```
  Just ts -> Just (dropComments ts)
```

```
where
```

```
  dropComments = filter notComment
```

```
  notComment (Comment _) = False
```

```
  notComment _ = True
```

Parsing Tokens

```
parseXml :: Parser Token Xml
parseXml = (token text >>= return . Txt) <|> do
  t <- token start
  ns <- many parseXml
  sat (isEnd t)
  return $ Xml t ns
where
  text (Text t) = Just t
  text _ = Nothing
  start (StartTag t) = Just t
  start _ = Nothing
  isEnd s (EndTag t) | s == t = True
  isEnd _ _ = False
```

Parsing Tokens

```
parseXml :: Parser Token Xml
parseXml = (token text >>= return . Txt) <|> do
  t <- token start
  ns <- many parseXml
  sat (isEnd t)
  return $ Xml t ns
where
  text (Text t) = Just t
  text _ = Nothing
  start (StartTag t) = Just t
  start _ = Nothing
  isEnd s (EndTag t) | s == t = True
  isEnd _ _ = False
```

```
fromString :: String -> Maybe Xml
fromString xs = case tokenize xs of
  Just ts -> parse p ts
  Nothing -> Nothing
  where p = do { xml <- parseXml; eoi; return xml }
```

Homework (for January 18th)

1. Read Chapter 10 of [Real World Haskell](#).
2. Write an `xmlToString` function, such that, for example `Xml "div" [Txt "test"]` results in something similar to `<div>test</div>`.
3. Implement a parser `balpar :: Parser Char Int` that accepts strings of balanced parentheses according to the grammar $\langle P \rangle ::= \langle P \rangle \langle P \rangle \mid (\langle P \rangle) \mid \epsilon$ and returns the total number of pairs of parentheses. However, note that implementing the above grammar directly leads to nontermination and is thus not an option.

Example: `ghci> parse balpar "()(()())"`
Just 6

4. Implement a parser `parseSet :: Parser Char (Set Int)` for sets of integers.

Example: `ghci> parse parseSet "{1,10,3,1}"`
Just {1,3,10}

Homework (for January 18th, continued)

5. Given the type of tokens

```
data T = LP | RP | DOT | LAM | VAR String
  deriving Show
```

write a lexer for lambda terms, that is, a parser

`lexTerm :: Parser Char [T]` that ignores white space and produces a list of tokens, where `LP` is a left parenthesis, `RP` a right parenthesis, `DOT` a dot, `LAM` a backslash, and `VAR` any sequence of characters excluding the preceding ones and white space.

Example: `ghci> parse lexTerm "\\x. (y"`
`Just [LAM,VAR "x",DOT,LP,VAR "y"]`

6. Using the type `T` of the previous exercise, implement a parser `parseTerm :: Parser T Term` for fully-parenthesized lambda terms as given by the grammar $t ::= x \mid (\backslash x.t) \mid (t t)$

Example: `ghci> parse parseTerm [LP,VAR "x",VAR "y",RP]`
`Just (x y)`