Computational Logic

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 9

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, implementing a type checker, 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, implementing a type checker, 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

Examples of Symbol Sequences

- text in natural language

What is Parsing?

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

Examples of Symbol Sequences

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

What is Parsing?

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

Examples of Symbol Sequences

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

What is Parsing?

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

Examples of Symbol Sequences

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

What is Parsing?

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

Examples of Symbol Sequences

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

What is Parsing?

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

Examples of Symbol Sequences

- 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)

$$
\begin{array}{rcl}
\langle \mathit{xml} \rangle & ::= & \texttt{<}\langle \mathit{name} \rangle\texttt{>}\langle \mathit{xml} \rangle^{*}\texttt{</}\langle \mathit{name} \rangle\texttt{>} \\
& | & \langle \mathit{text} \rangle \\
\langle \mathit{name} \rangle & ::= & (\langle \mathit{letter} \rangle \mid \texttt{\_})(\langle \mathit{letter} \rangle \mid \texttt{\_} \mid \langle \mathit{digit} \rangle)^{*} \\
\langle \mathit{letter} \rangle & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z} \\
\langle \mathit{digit} \rangle & ::= & 0 \mid \cdots \mid 9 \\
\langle \mathit{text} \rangle & ::= & (\text{“every symbol except for <”})^{+}
\end{array}
$$

BNF of XML Data (Simplified)

$$
\begin{array}{rcl}
\langle xml \rangle & ::= & \texttt{<}\langle name \rangle\texttt{>}\langle xml \rangle^{*}\texttt{</}\langle name \rangle\texttt{>} \\
& | & \langle text \rangle \\
\langle name \rangle & ::= & (\langle letter \rangle \mid \texttt{\_})(\langle letter \rangle \mid \texttt{\_} \mid \langle digit \rangle)^{*} \\
\langle letter \rangle & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z} \\
\langle digit \rangle & ::= & 0 \mid \cdots \mid 9 \\
\langle text \rangle & ::= & (\text{“every symbol except for <”})^{+}
\end{array}
$$

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] -> (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

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`
- errors are indicated by returning `Nothing` (no exact error message)

Lexing and Parsing

- traditionally parsing is split into 2 phases

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens
- white space characters and comments are often dropped at this stage

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens
- white space characters and comments are often dropped at this stage
- parsing: the actual parser works on list of tokens provided by lexer

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens
- white space characters and comments are often dropped at this stage
- parsing: the actual parser works on list of tokens provided by lexer
- typically produces structured data

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens
- white space characters and comments are often dropped at this stage
- parsing: the actual parser works on list of tokens provided by lexer
- typically produces structured data
- combinator parsers can be used for both stages

Tokens for XML Data

```haskell
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

document

```
<ul>
  <li>some thing</li>
  <li>another thing</li>
</ul>
```

is represented by

```
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)
```

- parsing letters and digits

```
letter = sat (`elem` (['a'..'z']++['A'..'Z']))
digit  = sat (`elem` ['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)
```

- parsing letters and digits

```
letter = sat (`elem` (['a'..'z']++['A'..'Z']))
digit  = sat (`elem` ['0'..'9'])
```

- 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)
```

- parsing letters and digits

```
letter = sat (`elem` (['a'..'z']++['A'..'Z']))
digit  = sat (`elem` ['0'..'9'])
```

- accepting/rejecting with respect to list of tokens

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

- parsing single white spaces

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

Primitive Parser Combinators – Choice

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

Primitive Parser Combinators – Choice

- (parser) combinator produces 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 produces 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 produces 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

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

- $\langle p \rangle ::= \texttt{ab}$

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

- $\langle p \rangle ::= \texttt{ab}$
- `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

- $\langle p \rangle ::= \texttt{ab}$
- `p = char 'a' `bind` \_ -> char 'b'`
- `char 'a' `bind` \x -> char 'b' `bind` \y -> return [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 (>>=)
- translation uses following equalities (from top to bottom):

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

$$\texttt{do \{ } x \texttt{ <- } m \texttt{; } M \texttt{ \}} = m \texttt{ >>= (\textbackslash} x \texttt{ -> do \{ } M \texttt{ \})}$$

$$\texttt{do \{ } m \texttt{; } M \texttt{ \}} = m \texttt{ >>= (\textbackslash\_ -> do \{ } M \texttt{ \})}$$

$$\texttt{do \{ } M \texttt{ \}} = M$$

Monads and do-Notation

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

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

$$\texttt{do \{ } x \texttt{ <- } m \texttt{; } M \texttt{ \}} = m \texttt{ >>= (\textbackslash} x \texttt{ -> do \{ } M \texttt{ \})}$$

$$\texttt{do \{ } m \texttt{; } M \texttt{ \}} = m \texttt{ >>= (\textbackslash\_ -> do \{ } M \texttt{ \})}$$

$$\texttt{do \{ } M \texttt{ \}} = M$$

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
```

Instantiating Type Classes

general schema for turning type `T` into instance of type class `C`

```
instance C T where
  -- implementations of class functions
```

Instantiating Type Classes

general schema for turning type `T` into instance of type class `C`

```
instance C T where
  -- implementations of class functions
```

Example – Equality for User-Defined Type

- consider type `data YNM = Yes | No | Maybe`
- instance declaration

```
instance Eq YNM where
  Yes   == Yes   = True
  No    == No    = True
  Maybe == Maybe = True
  _     ==  _    = False
```

Instantiating Type Classes

general schema for turning type `T` into instance of type class `C`

```
instance C T where
  -- implementations of class functions
```

Example – Equality for User-Defined Type

- consider type `data YNM = Yes | No | Maybe`
- instance declaration

```
instance Eq YNM where
  Yes   == Yes   = True
  No    == No    = True
  Maybe == Maybe = True
  _     ==  _    = False
```

Example – Parsers are Monads

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

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

- $\langle p \rangle ::= \mathtt{a}\langle p \rangle \mid \varepsilon$

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

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

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

- $\langle p \rangle ::= \mathrm{a}\langle p \rangle \mid \varepsilon$ (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

- $\langle p \rangle ::= \mathrm{a}\langle p \rangle \mid \varepsilon$ (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

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

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

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

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

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

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

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

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

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

```haskell
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 "-->")
  many space
  return $ Comment cmt

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

Lexing / Tokenization

```
lexer :: Parser Char [Token]
lexer = do
  many space
  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

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 }
```

Exercises (for January 12th)

1. Read Chapter 10 of Real World Haskell and prepare for the test.
2. Modify `lexer` to handle an optional XML prolog of the form `<?xml version="1.0" encoding="UTF-8"?>`.
3. Write your own `Show` instance for the data type `Xml`, such that, for example `Xml "div" [Txt "test"]` is printed as `<div>test</div>`.
4. Use the parsers and combinators from this lecture to define a function `uibkMail :: String -> Maybe (String, String)` that accepts an email address of the form ⟨*forename*⟩`.`⟨*surname*⟩`@student.uibk.ac.at` (where `student.` is optional) and returns the pair of forename and surname.
5. Implement a parser for comma-separated lists of integers. For example, running the parser on `"[0,1,3,3]some more text"` should result in `Just ([0,1,3,3], "some more text")`.
6. Implement a function `select : Tag -> Xml -> [Xml]` that gives the list of all XML nodes with a specific tag name. For example `select "li" (Xml "ul" [Xml "li" [Txt "a"], Xml "li" []])` should result in `[Xml "li" [Txt "a"], Xml "li" []]`.