Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

December 1, 2010

Today's Topics

- Parsing - Motivation
- Combinator Parsing
- Parsing Arithmetic Expressions

Parsing - Motivation

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language
- a computer program

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language
- a computer program
- a website

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language
- a computer program
- a website
- a piece of music

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language
- a computer program
- a website
- a piece of music
- a sequence of genes

What is Parsing

- parsing is the decomposition of a linear sequence into a structure, given by a grammar
- the linear sequence may be
- text in natural language
- a computer program
- a website
- a piece of music
- a sequence of genes
- ...

In the Following

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

In the Following

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

Notes

- BNF can express context-free grammars (CFGs)
- combinator parsers can parse context-sensitive grammars
- however, for this lecture, CFGs suffice

Example - CFG for Arithmetic Expressions

$$
\begin{array}{rcll}
\langle \mathit{expr} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{expr} \rangle + \langle \mathit{term} \rangle & \text{addition} \\
& \mid & \langle \mathit{expr} \rangle - \langle \mathit{term} \rangle & \text{subtraction} \\
& \mid & \langle \mathit{term} \rangle & \\
\\
\langle \mathit{term} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{term} \rangle * \langle \mathit{fact} \rangle & \text{multiplication} \\
& \mid & \langle \mathit{term} \rangle \,/\, \langle \mathit{fact} \rangle & \text{division} \\
& \mid & \langle \mathit{fact} \rangle & \\
\\
\langle \mathit{fact} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{num} \rangle & \\
& \mid & (\langle \mathit{expr} \rangle) & \\
& \mid & -\langle \mathit{fact} \rangle & \\
\\
\langle \mathit{num} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{digit} \rangle^{+} & \\
\\
\langle \mathit{digit} \rangle & \stackrel{\text{def}}{=} & 0 \mid \cdots \mid 9 &
\end{array}
$$

Example - Rewritten CFG (avoid Left Recursion)

$$
\begin{array}{rcl}
\langle \mathit{expr} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{term} \rangle \langle \mathit{expr'} \rangle \\
\langle \mathit{expr'} \rangle & \stackrel{\text{def}}{=} & +\langle \mathit{term} \rangle \langle \mathit{expr'} \rangle \\
 & | & -\langle \mathit{term} \rangle \langle \mathit{expr'} \rangle \\
 & | & \varepsilon \\
 & & \\
\langle \mathit{term} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{fact} \rangle \langle \mathit{term'} \rangle \\
\langle \mathit{term'} \rangle & \stackrel{\text{def}}{=} & *\langle \mathit{fact} \rangle \langle \mathit{term'} \rangle \\
 & | & /\langle \mathit{fact} \rangle \langle \mathit{term'} \rangle \\
 & | & \varepsilon \\
 & & \\
\langle \mathit{fact} \rangle & \stackrel{\text{def}}{=} & \langle \mathit{num} \rangle \\
 & | & (\langle \mathit{expr} \rangle) \\
 & | & -\langle \mathit{fact} \rangle \\
 & & \\
 & \vdots &
\end{array}
$$

Parsers - First Attempt

- functions of type `[t] -> (a, [t])`
- i.e., read some tokens from the given list, produce some result (of type `a`) together with the list of remaining tokens
- e.g., `digit "12"` results `('1', "2")`
- but what about errors? (e.g., `digit "abc"`)

Parsers - First Attempt

- functions of type `[t] -> (a, [t])`
- i.e., read some tokens from the given list, produce some result (of type `a`) together with the list of remaining tokens
- e.g., `digit "12"` results `('1', "2")`
- but what about errors? (e.g., `digit "abc"`)

Type of Parsers

- use **`newtype`** to distinguish from similar function types

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

- a 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 spaces and comments may be 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 spaces and comments may be 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 spaces and comments may be dropped at this stage
- parsing: the actual parser works on list of tokens provided by lexer
- produces an abstract syntax tree (AST)

Lexing and Parsing

- traditionally parsing is split into 2 phases
- lexing: divide original input (list of `Char`s) into other type of tokens
- white spaces and comments may be dropped at this stage
- parsing: the actual parser works on list of tokens provided by lexer
- produces an abstract syntax tree (AST)
- combinator parsers can be used for both stages

Tokens for Arithmetic Expressions

```
data Token = Lpar | Rpar
           | Plus | Minus
           | Star | Slash
           | Number Integer
  deriving (Show, Eq)
```

Tokens for Arithmetic Expressions

```
data Token = Lpar | Rpar
           | Plus | Minus
           | Star | Slash
           | Number Integer
  deriving (Show, Eq)
```

AST of Arithmetic Expressions

```
data Expr = Nat Integer
          | Neg Expr
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
  deriving Show
```

Combinator Parsing

Primitive Parsers

- only accept end of input

```
eoi :: Parser t ()
eoi = Parser (\ts ->
  case ts of []    -> Just ((),[])
             x:xs -> Nothing)
```

Primitive Parsers

- only accept end of input

```
eoi :: Parser t ()
eoi = Parser (\ts ->
  case ts of []   -> Just ((),[])
             x:xs -> Nothing)
```

- reading a single token

```
token :: (t -> Maybe a) -> Parser t a
token test = Parser (\ts ->
  case ts of
    []   -> Nothing
    x:xs ->
      case test x of
        Just y  -> Just (y, xs)
        Nothing -> Nothing)
```

Some Derived Parsers

- reading single characters

```
sat p    = token (\t -> if p t then Just t
                                else Nothing)
anyChar = sat (const True)
char c   = sat (==c)
```

Some Derived Parsers

- reading single characters

```
sat p   = token (\t -> if p t then Just t
                              else Nothing)
anyChar = sat (const True)
char c  = sat (==c)
```

- reading letters and digits

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

Some Derived Parsers

- reading single characters

```
sat p    = token (\t -> if p t then Just t
                                else Nothing)
anyChar = sat (const True)
char c   = sat (==c)
```

- reading letters and digits

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

- choosing from list of tokens

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

Some Derived Parsers

- reading single characters

```
sat p    = token (\t -> if p t then Just t
                                else Nothing)
anyChar = sat (const True)
char c   = sat (==c)
```

- reading letters and digits

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

- choosing from list of tokens

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

- parsing single white spaces

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

Turning Values into Parsers

- definition

```
lift :: a -> Parser t a
lift x = Parser (\ts -> Just (x,ts))
```

- `lift x` takes the value `x` and yields a parser that returns `x` without consuming any input

Parser Combinators - Sequencing Parsers

- 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)
```

Parser Combinators - Sequencing Parsers

- 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 2 arguments

Parser Combinators - Sequencing Parsers

- 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 2 arguments
- first a parser with results of type `a`

Parser Combinators - Sequencing Parsers

- 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 2 arguments
- first a parser with results of type `a`
- then, function taking `a` and producing a parser with results of type `b`

Parser Combinators - Sequencing Parsers

- 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 2 arguments
- first a parser with results of type `a`
- then, function taking `a` and producing a parser with results of type `b`
- `bind p f`, first executes `p` and then feeds the function `f` with its result

Parser Combinators - Sequencing Parsers

- 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 2 arguments
- first a parser with results of type `a`
- then, function taking `a` and producing a parser with results of type `b`
- `bind p f`, first executes `p` and then feeds the function `f` with its result
- since `f` is a function producing a parser, the result of `bind p f` is a parser

Parser Combinators - Choice between two Parsers

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

Parser Combinators - Choice between two Parsers

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

Example

- $\langle p \rangle \stackrel{\text{def}}{=} \mathtt{a} \mid \mathtt{b}$
- `p = char 'a' <|> char 'b'`
- i.e., `<|>` corresponds to $\mid$ in BNF

Parser Combinators - Iterate Parsers

- `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
- parsing sequences of white spaces

```
spaces = many space >> return ()
```

Parser Combinators - Iterate Parsers

- `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
- parsing sequences of white spaces

```
spaces = many space >> return ()
```

Example

- $\langle p \rangle \stackrel{\text{def}}{=} \mathsf{a}\langle p \rangle \mid \varepsilon$
- `p = many (char 'a')`
- $\langle p \rangle \stackrel{\text{def}}{=} \mathsf{a}\langle p \rangle \mid \mathsf{a}$
- `p = many1 (char 'a')`

Auxiliary Combinators

- apply a parser between to others

```
between ::
  Parser t a -> Parser t b -> Parser t c
    -> Parser t c
between l r p = l >> p >>= \x -> r >> return x
```

- apply a parser followed by another one

```
followedBy ::
  Parser t a -> Parser t b -> Parser t a
p `followedBy` q = do {x <- p; q; return x}
```

- in both cases we use the combinators, whenever we are not interested in the result of the last parser (`r` for `between` and `q` for `followedBy`)

Running Parsers on Input

- for testing purposes

```
test :: Parser t a -> [t] -> a
test p ts = case run p ts of
  Just (x, _) -> x
  Nothing     -> error "no parse"
```

Running Parsers on Input

- for testing purposes

```haskell
test :: Parser t a -> [t] -> a
test p ts = case run p ts of
  Just (x, _) -> x
  Nothing     -> error "no parse"
```

- applying a parser to a list of tokens

```haskell
parse :: Parser t a -> [t] -> Maybe a
parse p ts = case run p ts of
  Just (x, _) -> Just x
  Nothing     -> Nothing
```

Do-Notation for Parsers

- parsers are very similar to IO actions
- instead of reading input and writing output, parsers read tokens and store the remaining tokens
- as for IO actions, parsers can be run in sequence, and arbitrary values can be turned into parsers using `lift`
- this pattern is so common that there is a dedicated type class

Do-Notation for Parsers

- parsers are very similar to IO actions
- instead of reading input and writing output, parsers read tokens and store the remaining tokens
- as for IO actions, parsers can be run in sequence, and arbitrary values can be turned into parsers using `lift`
- this pattern is so common that there is a dedicated type class

The `Monad` Class - Supporting Do-Notation

- specification, all of:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

- `return` lifts an arbitrary value into a monad
- `(>>=)` (called 'bind'), executes two monads in a row, where the second may depend on the 'output' of the first

Monads and Do-Notation

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

$$\begin{aligned}
\texttt{do \{let } x \texttt{ = } e\texttt{;} M\texttt{\}} &= \texttt{let } x \texttt{ = } e \texttt{ in do \{} M \texttt{\}} \\
\texttt{do \{} x \texttt{ <- } m\texttt{;} M\texttt{\}} &= m \texttt{ >>= (\textbackslash} x \texttt{ -> do \{} M \texttt{\})} \\
\texttt{do \{} m\texttt{;} M\texttt{\}} &= m \texttt{ >>= (\textbackslash_ -> do \{} M \texttt{\})} \\
\texttt{do \{} M \texttt{\}} &= M
\end{aligned}$$

Example - IO

- the 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 scheme for turning type `T` into instance of type class `C`

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

Instantiating Type Classes

- general scheme 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 the 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 scheme 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 the 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
```

Parsing Arithmetic Expressions

Reading Tokens

- ignore trailing white space

```
lex p = p `followedBy` spaces
```

- reading tokens of type `Token`

```
lpar  = lex (char '(') >> return Lpar
rpar  = lex (char ')') >> return Rpar
plus  = lex (char '+') >> return Plus
minus = lex (char '-') >> return Minus
star  = lex (char '*') >> return Star
slash = lex (char '/') >> return Slash
num   =
  lex (many1 digit) >>= return . Number . read
```

- lexing the input (i.e., turn list of `Char`s into list of `Token`s)

```
tokenize = spaces >> many token
  where token = lpar <|> rpar <|> plus
            <|> minus <|> star <|> slash <|> num
```

Recognizing Tokens

```
nat   = token (\t ->
  case t of Lex.Number i -> Just (Nat i)
            _            -> Nothing)

justIf :: (a -> Bool) -> a -> Maybe ()
justIf p x = if p x then Just ()
                    else Nothing

lpar  = token (justIf (== Lex.Lpar))
rpar  = token (justIf (== Lex.Rpar))
plus  = token (justIf (== Lex.Plus))
minus = token (justIf (== Lex.Minus))
star  = token (justIf (== Lex.Star))
slash = token (justIf (== Lex.Slash))
```

Parsing Tokens

```
expr = term >>= expr'
  where
    expr' t = add <|> sub <|> return t
      where
        add = plus  >> term >>= expr' . Add t
        sub = minus >> term >>= expr' . Sub t

term = factor >>= term'
  where
    term' f = mul <|> div <|> return f
      where
        mul = star  >> factor >>= term' . Mul f
        div = slash >> factor >>= term' . Div f

factor = nat <|> par <|> neg
  where
    par = between lpar rpar expr
    neg = minus >> factor >>= return . Neg
```

Exercises (for December 10th)

1. Read chapter 10 of Real World Haskell
2. Write your own `Eq` instance for the data type `Term` from the lecture slides.
3. Write your own `Show` instance for the data type `Term` from the lecture slides.
4. Implement a function `eval :: Exp -> Integer`, computing the result of a given expression.
5. 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.
6. Implement a function `fromHex : String -> Maybe Int` that takes a string representation of a hexadecimal number and returns its decimal value as integer.