Parser

Thorben Gülck
Mathias Leonhardt

17. Dezember 2010

Inhaltsübersicht

Einleitung

Einleitung

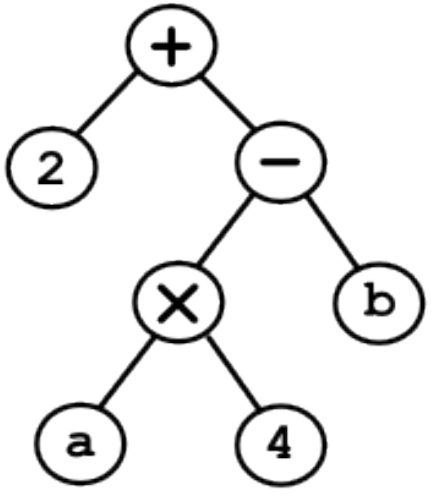
- Ein Parser nimmt eine Folge von Zeichen und produziert eine Datenstruktur, die der Interpretation der Zeichenfolge entspricht.
- Beispiel: $2 + a * 4 - b$

Einleitung

- Ein Parser nimmt eine Folge von Zeichen und produziert eine Datenstruktur, die der Interpretation der Zeichenfolge entspricht.
- Beispiel: $2 + a * 4 - b$

Einleitung

- Ein Parser nimmt eine Folge von Zeichen und produziert eine Datenstruktur, die der Interpretation der Zeichenfolge entspricht.
- Beispiel: $2 + a * 4 - b$



Weitere Anwendungsbeispiele für Parser

- Browser: müssen HTML Seite in DOM-Struktur umwandeln um sie anzuzeigen
- Interpretierer: muss ein Programm parsen um es auszuführen
- ...

Der Parser-Datentyp

```
-- Erster Ansatz

newtype Parser = MkParser(String -> Tree)
```

```
-- Moeglicherweise verarbeitet der Parser
-- die Eingabe nicht vollstaendig:
-- Resultat enthaelt Reststring

newtype Parser = MkParser(String -> (Tree, String))
```

```
-- deterministischer Parser:
--   Leere Liste: Fehler
--   Einelementige Liste: Erfolg

-- nichtdeterministischer Parser:
--   Liste ermoeglicht Erweiterung auf
--   mehrere Ergebniswerte, falls Eingabestring
--   auf mehrere Arten interpretierbar

newtype Parser = MkParser(String -> [(Tree, String)])
```

```
-- Erweiterung auf beliebige Ergebnistypen

-- Beispiele:

-- Parser fuer arithmetische
-- Ausdruecke liefert Baum

-- Parser fuer Zahlen liefert Integer

newtype Parser a = MkParser(String -> [(a, String)])

-- Funktion zur Anwendung eines Parsers auf eine Eingabe

apply :: Parser a -> String -> [(a, String)]
apply (MkParser p) input = p input
```

Die Parser-Monade

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

instance Monad Parser where

  return result = MkParser parser where
    parser input = [(result, input)]

  p >>= qfunc = MkParser parser where
    parser input =
      [(result2, rest2) |
       (result1, rest1) <- apply p input,
       (result2, rest2) <- apply (qfunc result1) rest1]
```

```haskell
-- erstes Zeichen lesen und als Ergebnis
-- zurueckgeben

item :: Parser Char
item = MkParser parser where
       parser [] = []
       parser (c:cs) = [(c,cs)]

-- Beispiele:

-- apply item ""
-- []

-- apply item "abc"
-- [('a',"bc")]
```

```haskell
-- Verkettung mehrerer item-Parser
-- Lesen des 1. und 3. Zeichen einer Eingabe

read_1_3 :: Parser (Char, Char)
read_1_3 = do
           x <- item
           item -- verwerfen des zweiten Zeichens
           y <- item
           return (x, y)

-- Beispiele:

-- apply read_1_3 "abcdef"
-- [('a', 'c'), "def"]

-- apply read_1_3 "ab"
-- []
```

```
-- Neutrales Element bzgl. >>=: return

-- Nullelement bzgl. >>= (schlaegt immer fehl):

zero :: Parser a
zero = MkParser parser where
       parser input = []

-- Anwendungsbeispiel fuer item und zero:

sat :: (Char -> Bool) -> Parser Char
sat pred = do
           c <- item
           if pred c then return c else zero
```

Basis-Parser

```haskell
-- Ueberlesen eines Zeichens
char :: Char -> Parser ()
char x = do
         sat (==x)
         return ()

-- Beispiele:

-- apply (char 'a') "abc"
-- [((), "bc")]

-- apply (char 'a') "def"
-- []
```

```haskell
-- Ueberlesen einer Zeichenfolge
string :: String -> Parser ()
string [] = return ()
string (x:xs) = do
                char x
                string xs
                return ()

-- Beispiele:

-- apply (string "abc") "abcdef"
-- [((), "def")]

-- apply (string "abc") "aaadef"
-- []
```

```haskell
lower :: Parser Char
lower = sat isLower

-- Beispiele:

-- apply lower "abc"
-- [('a', "bc")]

-- apply lower "Abc"
-- []

-- upper, letter, alphanum entsprechend
```

```haskell
digit :: Parser Int
digit = do
        d <- sat isDigit
        return (ord d - ord '0')

-- Beispiel:

-- apply digit "123"
-- [(1, "23")]

-- apply digit "abc"
-- []

addition :: Parser Int
addition = do
           m <- digit
           char '+'
           n <- digit
           return (m + n)
```

Alternative

```
plus :: Parser a -> Parser a -> Parser a
p `plus` q = MkParser parser where
             parser input = apply p input ++ apply q input

-- plus bildet Monoid mit zero als neutralem Element
-- zero `plus` p = p
-- p `plus` zero = p
-- p `plus` (q `plus` r) = (p `plus` q) `plus` r

-- Weiterhin gilt das Distributivgesetz:
-- (p `plus` q) >>= r  = (p >>= r) `plus` (q >>= r)
```

```
lowers :: Parser String
lowers = (do
           c <- lower
           cs <- lowers
           return (c:cs)) `plus` return ""

-- lowers "isUpper"
-- [("is", "Upper"), ("i", "sUpper"), ("", "isUpper")]
```

```
-- deterministische Version einer Alternative
-- (falls p und q deterministisch):

orelse :: Parser a -> Parser a -> Parser a
p `orelse` q = MkParser parser where
               parser input = case apply p input of
                                [] -> apply q input
                                res -> res

-- Vorsicht mit orelse!
-- Einfache arithmetische Ausdruecke: 1, 1+2
-- expr = digit `orelse` addition ?
-- expr = addition `orelse` digit ?
-- -> Faktorisieren!

-- Hinweise zu Linksrekursion:
-- ~si/vorlesungen/cb/SyntaxAnalyse/LinksRekursion.html
```

Wiederholung

```haskell
-- 0..n
many :: Parser a -> Parser [a]
many parser = (do
               x <- parser
               xs <- many parser
               return (x:xs)) `orelse` return []

-- 1..n
some :: Parser a -> Parser [a]
some parser = do
              x <- parser
              xs <- many parser
              return (x:xs)
```

```haskell
-- Beispiel: Bezeichner parsen
ident :: Parser String
ident = do
        c <- lower
        cs <- many alphanum
        return (c:cs)

-- Beispiel: Natuerliche Zahl parsen
nat :: Parser Int
nat = do
      digits <- some digit
      return (foldl1 (+) digits)
      where
      m + n = 10*m + n
```

```haskell
-- Beispiel: Ganze Zahl parsen
int :: Parser Int
int = (do
        char '-'
        n <- nat
        return (-n)) `orelse` nat

-- Beispiel: Liste ganzer Zahlen wie: [1,2,3]
ints :: Parser [Int]
ints = do
        char '['
        n <- int
        ns <- many (do {char ','; int})
        char ']'
        return (n:ns)
```

```haskell
-- Abstraktion: Wiederholung mit Trennzeichen
somewith :: Parser b -> Parser a -> Parser [a]
somewith q p = do
               x <- p
               xs <- many (do {q; p})
               return (x:xs)

-- Analog zu many:
manywith :: Parser b -> Parser a -> Parser [a]
manywith q p = somewith q p `orelse` return []
```

```haskell
space :: Parser ()
space = do
        many (sat isSpace)
        return ()

token :: Parser a -> Parser a
token parser = do
               space
               x <- parser
               space
               return x

symbol :: String -> Parser ()
symbol xs = token (string xs)
```

Arithmetische Ausdrücke

```
expr :: Parser Int
expr =  do
        t <- term
        (do
         symbol "+"
         e <- expr
         return (t+e)) `orelse` return t

term :: Parser Int
term =  do
        f <- factor
        (do
         symbol "*"
         t <- term
         return (f*t)) `orelse` return f
```

```
factor :: Parser Int
factor =  (do
           symbol "("
           e <- expr
           symbol ")"
           return e) `orelse` nat

eval :: String -> Int
eval input =  case (apply expr input) of
              [(n,[])]  -> n
              [(_,out)] -> error ("unused input " ++ out)
              []        -> error "invalid input"
```

Parsec

Parsec

- Parsec: **parse**r **c**ombinator library
- Stellt vorgestellte Basisfunktionalität zur Verfügung (uvm.)
- Fehlerkonzept: Position, unerwartete Eingabe, etc.
- Keine Spezialsyntax erforderlich

Parsec

- Benötigt libghc6-parsec-dev oder Haskell Platform auf Ubuntu
- Verwendung durch „import Parsec“
- Tutorial und Dokumentation:
  http://legacy.cs.uu.nl/daan/download/parsec/parsec.html
- Es existieren Clones für Java, C++, Erlang, Python, ...
  http://www.haskell.org/haskellwiki/Parsec

```haskell
-- data  Either a b  =  Left a
--                   | Right b

numbers = commaSep integer

-- parse :: Parser a -> FilePath -> String
--          -> Either ParseError a

main = case (parse numbers "" "1, 2, 3") of
       Left err -> print err
       Right xs -> print (sum xs)
```

```
-- Alternative: <|>

identifier = do
             c  <- letter
             cs <- many (alphaNum <|> char '_')
             return (c:cs)
```

Fragen?