Funktionale Programmierung
Masterstudiengang Informatik
Wintersemester 2010
Cornelius Schmale
Olaf Neumann

Funktionale Parser in Haskell

Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

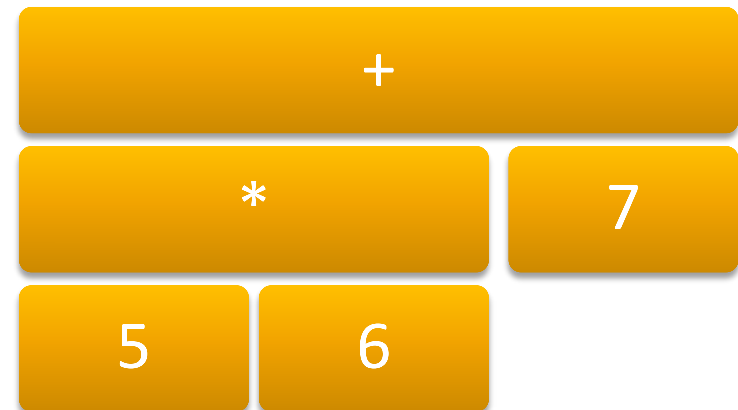
Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Definition Parser

Ein Parser ist ein Programm, dass eine Menge unstrukturierter Daten (Input) analysiert und seine syntaktische Struktur feststellt.

5*6+7



Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Der Typ *Parser*

Ein Parser benötigt einen String und erzeugt einen Baum:

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

Jedoch konsumiert ein Parser nicht seinen gesamten Input, sondern nur ein Teil davon und gibt den Rest zurück:

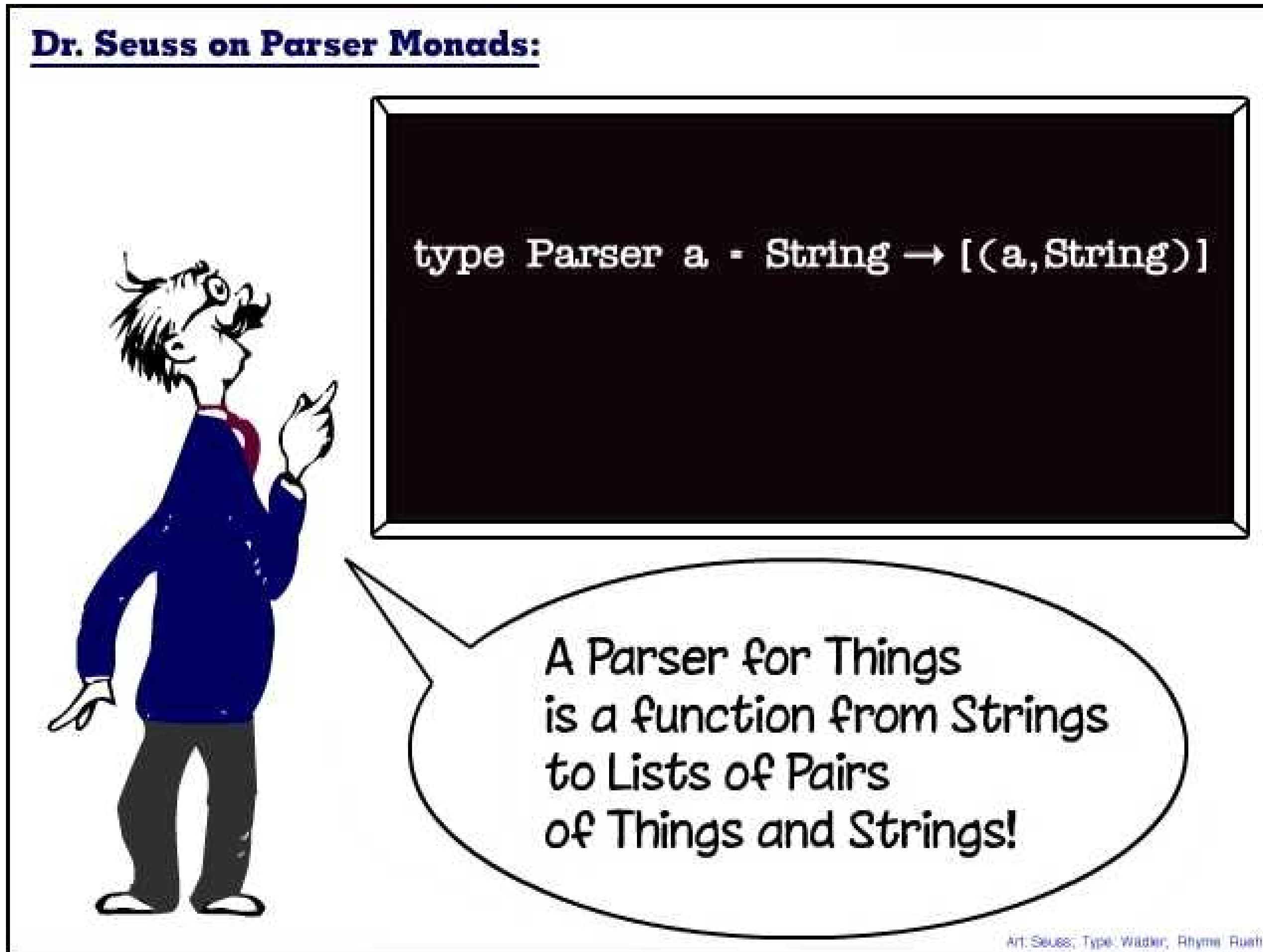
```
newtype Parser   = MkParser (String -> (Tree, String))
```

Ebenso kann das Parsen eines Ausdrucks fehlschlagen:

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

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

Der Typ Parser



Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Basisparser

Definitionen

`return v`
Hat immer Erfolg und liefert das Resultat value v und beachtet den Input nicht

`failure`
Schlägt immer fehl, beachtet den Input nicht

`item`
Schlägt fehl, wenn Input leer ist, sonst liefert er den ersten Char des Inputstrings

Basisparser

Implementierung

```
return          :: a -> Parser a
return v         = MkParser (\ inp -> [(v, inp)])

failure         :: Parser a
failure          = MkParser (\ inp -> [])

item            :: Parser Char
item             = MkParser (\ inp -> case inp of
                                        []     -> []
                                        (c:cs) -> [(c,cs)]
                            )
```

Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Parserkombination

Sequenz

Die einfachste Verschachtelung von Parsern ist das sequentielle Ausführen, bei dem der Output des Parsers_n der Input des Parsers_{n+1} ist.

```
instance Monad Parser where
  return x = MkParser (\ input -> [(x, input)])
  p >>= f  = MkParser (\ input -> case parse p input of
                                     []      -> []
                                     [(v,o)] -> parse (f v) o
                        )
```

```
parser1 >>= \value1 ->
parser2 >>= \value2 ->
…
parsern >>= \valuen ->
return (f value1 value2 … valuen)
```

```
do value1 <- parser1
   value2 <- parser2
   …
   valuen <- parsern
   return (f value1 value2 .. valuen)
```

Parserkombination

OrElse

Ausführung des ersten Parsers mit dem Inputstring, wenn dies fehlschlägt den zweiten Parser mit dem Inputstring ausführen.

```
orelse     :: Parser a -> Parser a -> Parser a
orelse p q = MkParser (\ i -> case parse p i of
                                   []          -> parse q i
                                   [(v, out)] -> [(v, out)]
                      )
```

Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Abgeleitete primitive Parser

sat

Durch Kombination der 3 Basisparser *item*, *return* und *failure* mit Sequenz und *OrElse* können weitere nützliche primitive Parser definiert werden.

Zuerst definieren wir einen Parser *sat* für einen einzelnen Char, der ein Prädikat p erfüllt:

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

Abgeleitete primitive Parser

digit, lower, upper, letter, alphanum

Mit dem Charparser sat können nun weitere Parser definiert werden:

```
digit    = sat isDigit
lower    = sat isLower
upper    = sat isUpper
letter   = sat isAlpha
alphanum = sat isAlphaNum

char    :: Char -> Parser Char
char c   = sat (==c)
```

Abgeleitete primitive Parser

string, many, atLeastOne

```
-- a parser for a certain string
string        :: String -> Parser String
string []      = return []
string (c:cs) = do char c
                   string cs
                   return (c:cs)

-- repeat a parser
many   :: Parser a -> Parser [a]
many p = atLeastOne p `orelse` return []

-- repeat at least once
atLeastOne   :: Parser a -> Parser [a]
atLeastOne p = do x  <- p
                  xs <- many p
                  return (x:xs)
```

Abgeleitete primitive Parser

space, nat

```
-- a whitespace parser
space :: Parser ()
space   = do many (sat isSpace)
             return ()

-- a natural number parser
nat :: Parser Int
nat   = do n <- atLeastOne digit
           return (read n)
```

Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Behandlung von Whitespace

```haskell
-- a parser to discard spaces
token   :: Parser a -> Parser a
token p = do space
             r <- p
             space
             return r

-- a parser for a certain symbol discarding spaces
symbol     :: String -> Parser String
symbol sym = token (string sym)

-- a numeric constant parser
natConst :: Parser Expr
natConst   = do i <- token nat
               return (Const i)
```

Agenda

- Definition
- Typdefinition
- Basisparser
- Parserkombinationen
- Abgeleitete primitive Parser
- Whitespacebehandlung
- Beispielparser für einen *simple expression evaluator*

Arithmetische Ausdrücke

Eine Grammatik

Ein Ausdruck aus Zahlen, +, -, *, /, %, (,)
Operationen sind rechtsassoziativ: 3 + 4 + 5 -6 bedeutet 3 + (4 + (5 -6))

*, / und % haben eine höhere Priorität als + und –
3 – 4 * 5 bedeutet 3 – (4 * 5)

```
expr ::= expr + expr
       | expr – expr
       | expr * expr
       | expr / expr
       | expr % expr
       | (expr)
       | nat
```

```
nat ::= 0
      | 1
      | 2
      | …
```

```
expr ::= expr (+|-|*|/|%) expr
       | (expr)
       | nat
```

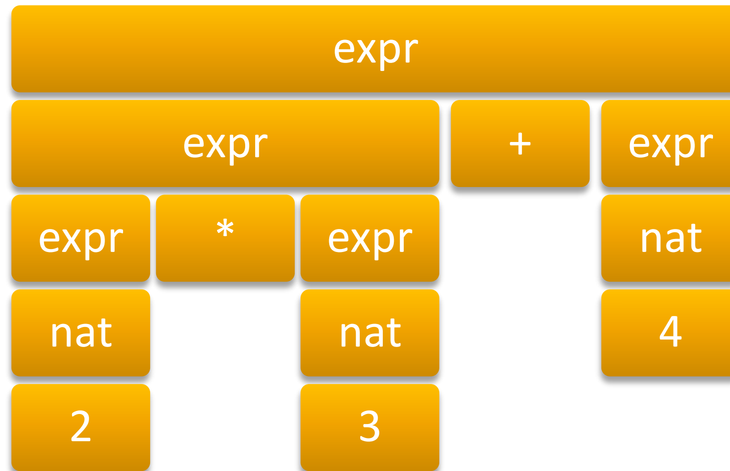
Arithmetische Ausdrücke

Eine Grammatik

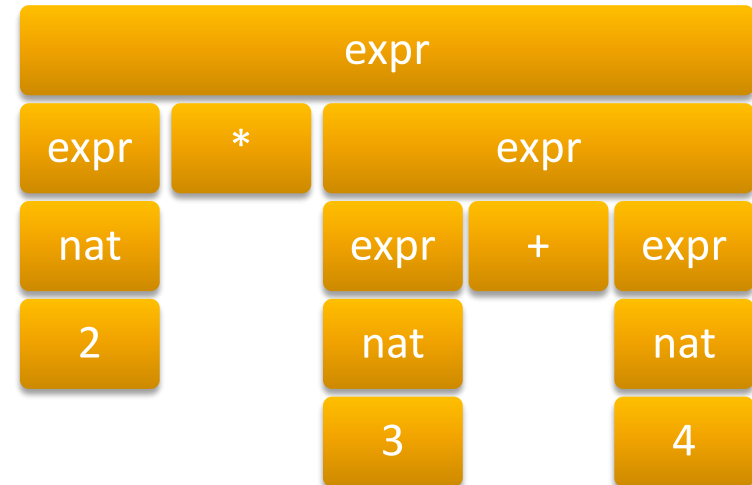
```
nat  ::= 0 | 1 | 2 …
expr ::= expr (+|-|*|/|%) expr
       | (expr)
       | nat
```

Durch Anwendung der Grammatik kann die Konstruktion des Ausdruckes 2*3+4 durch folgende Ableitungsbäume dargestellt werden:

(2 * 3) + 4



2 * (3 + 4)

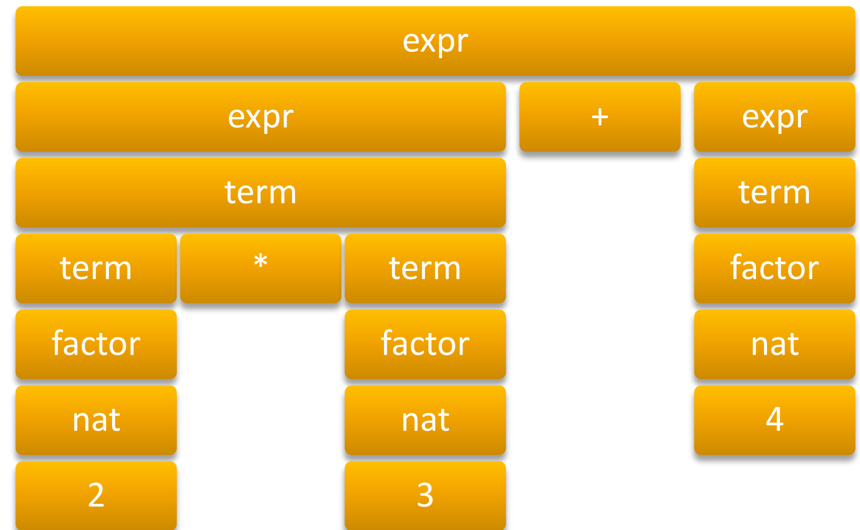


Diese Grammatik kann keine Operaterprioritäten darstellen!

Arithmetische Ausdrücke

Priorität der Operatoren der Grammatik

```
expr     ::= expr (+|-) expr
           | term

term     ::= term (*|/|%) term
           | factor

factor   ::= (expr)
           | nat

nat      ::= 0 | 1 | 2 | …
```

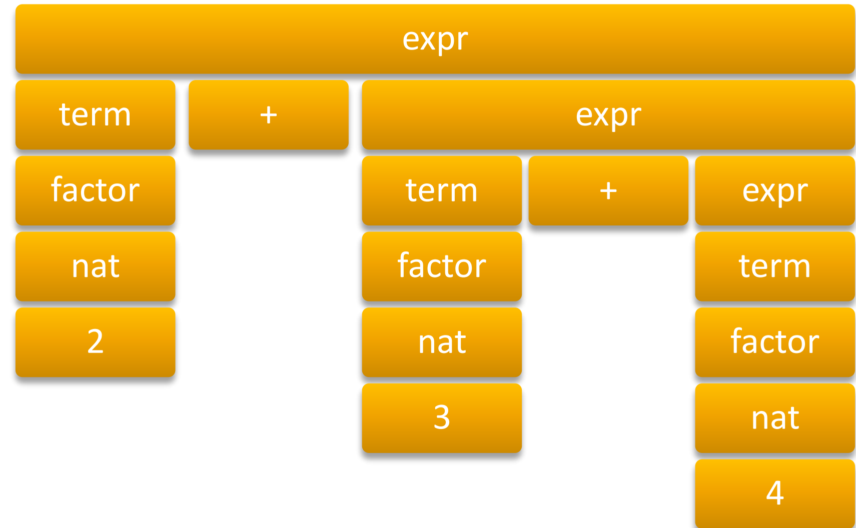


Prioritäten jetzt richtig, aber Rechtsassoziativität nicht beachtet:
Der Ausdruck 2+3+4 hat 2 mögliche Ableitungsbäume, (2+3)+4 und 2+(3+4).
Daher weitere Änderung an den beiden Ableitungsregeln:

Arithmetische Ausdrücke

Rechtsassoziativität der Grammatik

```
expr    ::= term (+|-) expr
          | term

term    ::= factor (*|/|%) term
          | factor

factor  ::= (expr)
          | nat

nat     ::= 0 | 1 | 2 | …
```



Jetzt hat 2+3+4 nur noch einen Ableitungsbaum, nämlich 2+(3+4) und ist somit eindeutig:
Jeder wohlgeformte Ausdruck hat nur einen einzigen Ableitungsbaum.

Arithmetische Ausdrücke

Letzte Optimierung der Grammatik

```
expr    ::= term (+|-) expr
             | term
term    ::= factor (*|/|%) term
             | factor
factor  ::= (expr)
             | nat
nat     ::= 0 | 1 | 2 | …
```

```
expr    ::= term ((+|-) expr | ε )
term    ::= factor ((*|/|%) term | ε )
factor  ::= (expr) | nat
nat     ::= 0 | 1 | 2 | …
```

Simple Expression Evaluator

Erinnerung

```
data Expr  = Const  Int
           | Binary BinOp Expr Expr
             deriving (Show)

data BinOp = Add
           | Sub
           | Mul
           | Div
           | Mod
             deriving (Eq, Show)
```

Arithmetische Ausdrücke

Umsetzen der Grammatik in einzelne Parser - Operatoren

```
data BinOp = Add
           | Sub
           | Mul
           | Div
           | Mod
```

Zunächst Parser für Operatoren:

```
binOpAddSub :: Parser BinOp
binOpAddSub  = do op <- symbol "+" `orelse` symbol "-"
                  case op of
                    "+" -> return Add
                    "-" -> return Sub

binOpMulDiv :: Parser BinOp
binOpMulDiv  = do op <- symbol "*" `orelse` symbol "/" `orelse` symbol "%"
                  case op of
                    "*" -> return Mul
                    "/" -> return Div
                    "%" -> return Mod
```

Arithmetische Ausdrücke

Umsetzen der Grammatik in einzelne Parser - Elemente

```
expr :: Parser Expr
expr  = do t <- term
           do o <- binOpAddSub
              e <- expr
              return (Binary o t e)
              `orelse` return t

term :: Parser Expr
term  = do f <- factor
           do o <- binOpMulDiv
              t <-term
              return (Binary o f t)
              `orelse` return f

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

```
expr    ::= term ((+|-) expr | ε )
term    ::= factor ((*|/|%) term |ε )
factor  ::= (expr) | nat
nat     ::= 0 | 1 | 2 | …

data Expr  = Const  Int
           | Binary BinOp Expr Expr
             deriving (Show)
```

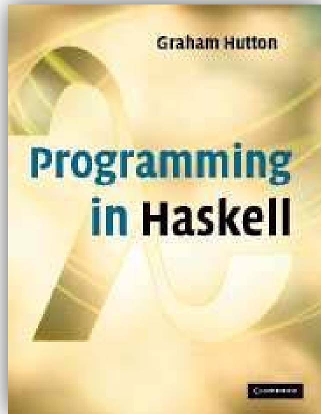
Simple Expression Parser

Zusammenführung

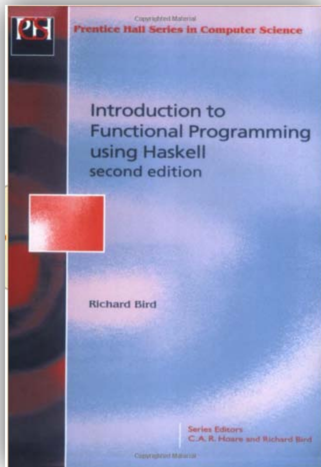
```
doParse    :: String -> Expr
doParse s   = case parse expr s of
                 [(n,[])]  -> n
                 [(_,out)] -> error ("unused input " ++ out)
                 []        -> error "invalid input"

eval :: Expr -> Result Int

doEval :: String -> Result Int
doEval  = eval . doParse
```

Literaturangaben

Bücher, Videos und weitere Quellen



Programming in Haskell
Graham Hutton
ISBN: 978-0-521-69269-4
http://cambridge.org/9780521692694



Introduction to Functional Programming
Richard Bird
ISBN: 978-0-134-84346-9

Simple Expression Evaluator
Prof. Dr. Uwe Schmidt
www.fh-wedel.de/~si/vorlesungen/fp/Monaden/Expr1.hs

Functional Parsers in Haskell

Vielen Dank für Ihre Aufmerksamkeit

Fragen?
Anregungen?
Kommentare?