Funktionale Parser

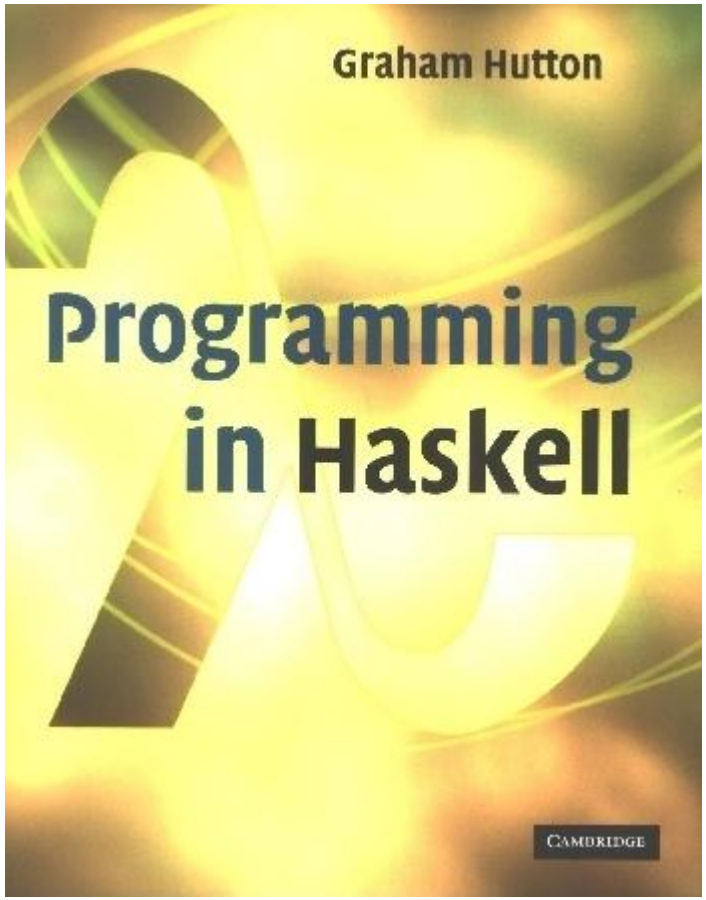
Tobias Meyn & Thiemo Alldieck

Literatur

Graham Hutton
University of Nottingham

Cambridge University Press
Erschienen: 2007

Preis: ~30,00€

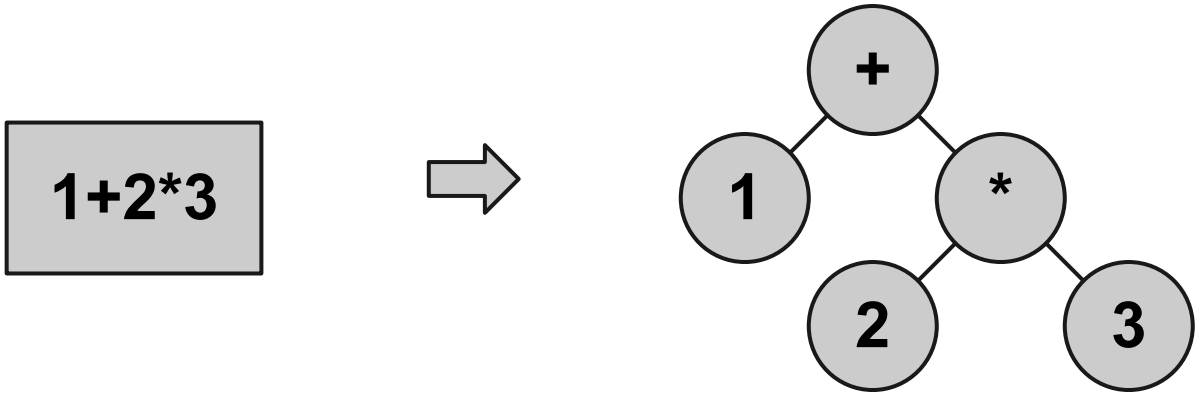


Gliederung

- Der Parser-Typ
- Grundlegende Parser
- Sequencing
- Primitive
- Choice
- Repetition
- Parser für Listen Versuch 1
- Whitespace
- Parser für Listen Versuch 2
- Parser für Arithmetische Ausdrücke

Was sind Parser?

- Umwandeln eines Strings in einen Syntax-Baum.



Der Parser-Typ

- Entwicklung des Parser-Typ
  - String in Tree umwandeln.

```
type Parser  :: ??
```

Der Parser-Typ

- Entwicklung des Parser-Typ
  - String in Tree umwandeln.

```
type Parser  :: String -> Tree
```

Der Parser-Typ

- Entwicklung des Parser-Typ
  - String in Tree umwandeln.

```
type Parser  :: String -> Tree
```

- Problem:
  - Annahme: ganzer String wird ausgewertet.
  - Fall: Parser für Nummern bei Argument `"123abc"`

Der Parser-Typ

- Entwicklung des Parser-Typ
  - Nicht verwerteten Teil zurückgeben.

```
type Parser  :: String -> (Tree, String)
```

Der Parser-Typ

- Entwicklung des Parser-Typ
  - Nicht verwerteten Teil zurückgeben.

```
type Parser  :: String -> (Tree, String)
```

- Problem:
  - Annahme: Parser immer erfolgreich
  - Fall: Parser für Nummern bei Argument `"abc"`

Der Parser-Typ

- Entwicklung des Parser-Typ
  - Beachtung des Fehlerfalls (=leere Liste).

```
type Parser  :: String -> [(Tree, String)]
```

Der Parser-Typ

- Entwicklung des Parser-Typ
  - Beachtung des Fehlerfalls (=leere Liste).

```
type Parser  :: String -> [(Tree, String)]
```

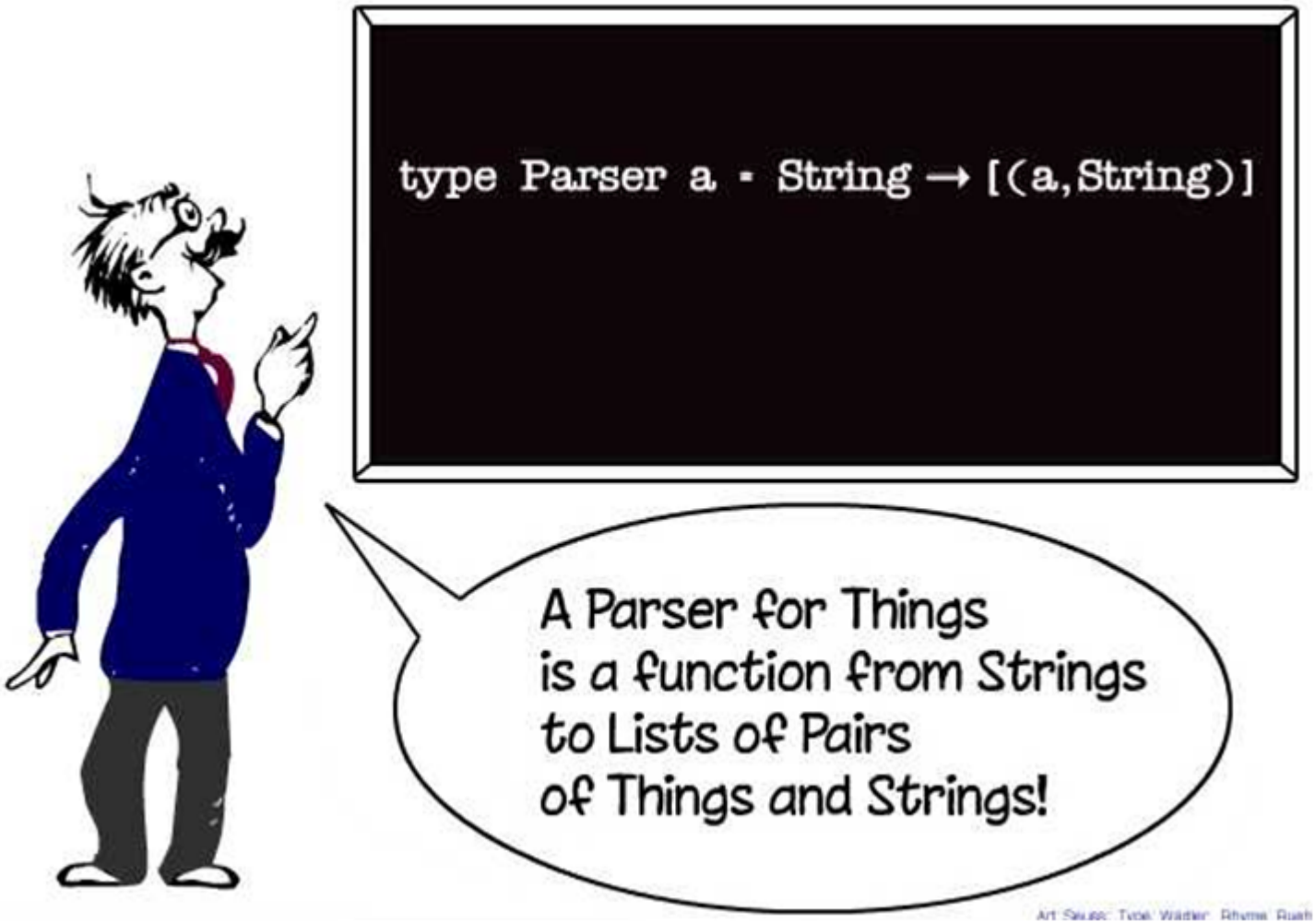
- Problem:
  - Spezifischer Typ: Tree
  - Letzter Schritt: Abstrahierung des Parser Typs.

Der Parser-Typ

- Entwicklung des Parser-Typ
  - Abstrahierung des Parser Typs.

```
type Parser a :: String -> [(a, String)]
```

Der Parser-Typ



Der Parser-Typ

Etwas schöner:

```
type Result a = Maybe (a, String)
data Parser a = Parser (String -> Result a)
```

Warum? Später mehr...

Quiz

- Absicht:
  Entwicklung der grundlegenden Bestandteile.

- Ideen?

Tipp: Aller guten Dinge sind 3.

Grundlegende Parser

- Grundlegende Parser definiert.
- Anwendung in der Folge mit Funktion `parse`

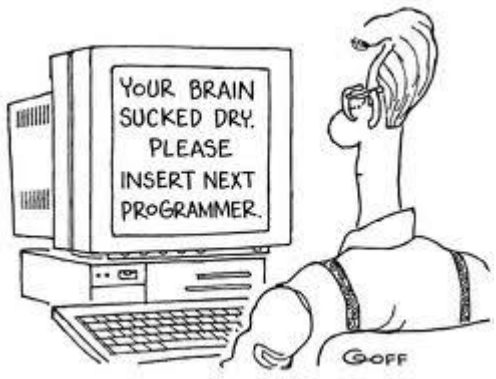
```
parse :: Parser a -> String -> Result a

parse (Parser p) inp :: p inp
```

Grundlegende Parser

Ergebnis?

```
parse (return 1) "123"
parse failure "abc"
parse item ""
parse item "abc"
```



Quiz

Wie wende ich `item` 2x an?

Wo liegt das Problem?

Lösung?

Sequencing

Monaden!

```
instance Monad Parser where
  return v    = Parser $ \inp -> Just (v, inp)
  p >>= f     = Parser $
                    \inp -> case parse p inp of
                        Nothing          -> Nothing
                        Just (v, out)    -> parse (f v) out
```

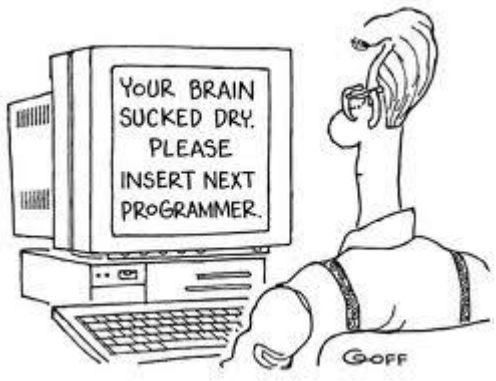
Sequencing

Beispiel:

```
psequence    :: Parser (Char, Char)
psequence    =  do
                  x <- item
                  item
                  y <- item
                  return (x,y)
```

Ergebnis?

```
parse psequence "abc"
parse psequence "ab"
```



Primitive

Grundlage: Satisfaction

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

Primitive

```
char  :: Char -> Parser Char

char x = ???
```

Primitive

```
char  :: Char -> Parser Char

char x = sat (==x)
```

Primitive

```
digit    :: Parser Char
digit    =  sat isDigit

lower :: Parser Char
lower    =  sat isLower

upper    :: Parser Char
upper    =  sat isUpper
```

Primitive

```
letter  :: Parser Char
letter  =  sat isAlpha

alphanum :: Parser Char
alphanum =  sat isAlphaNum
```

Primitive

```
string          :: String -> Parser String

string []       = ???

string (x:xs)   = ???
```

Primitive

```
string          :: String -> Parser String

string []       = return []

string (x:xs) = ???
```

Primitive

```
string          :: String -> Parser String

string []       = return []

string (x:xs) = do
                    char x
                    string xs
                    return (x:xs)
```

Quiz

Wie parse ich einen `string` ***oder*** eine `digit`?

Problem?

Lösung?

Choice

Wende p an, wenn p fehlschlägt wende q an.

```
(+++)      :: Parser a -> Parser a -> Parser
a
p +++ q =  Parser $
             \inp ->
               case parse p inp of
                    Nothing   -> parse q inp
                    Just x    -> Just x
```

(Oder `instance MonadPlus Parser`)

Choice

Ergebnis?

```
parse (digit +++ lower) "1"
parse (digit +++ lower) "a"
parse (digit +++ lower) "A"
parse (digit +++ lower) "."
```

Quiz

Wie parse ich eine natürliche Zahl?

1, 12, 34, 42, 666, ...

Repetition

```
many    :: Parser a -> Parser [a]
many p  =  ???

many1     :: Parser a -> Parser [a]
many1 p =  ???
```

Tipp: Man hilft sich gegenseitig.

Repetition

```
many    :: Parser a -> Parser [a]
many p  =  many1 p +++ return []

many1      :: Parser a -> Parser [a]
many1 p =  do
             v <- p
             vs <- many p
             return (v:vs)
```

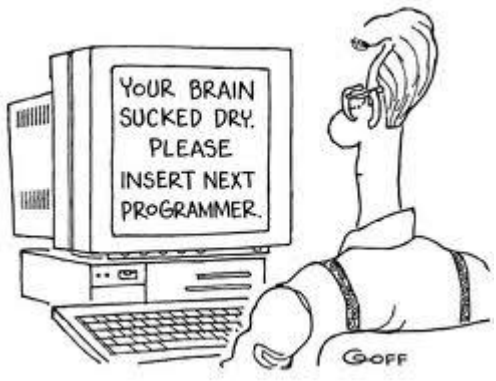
Repetition

```
nat     :: Parser Int
nat     =  do
             xs <- many1 digit
             return (read xs)
```

Ergebnis?

```
parse nat "42"
parse nat "42a"
```

```
read :: Read a => String -> a
```



Parser für Listen

```
plist1    :: Parser [Int]
plist1    =  do
               ???
```

Für nicht-leere Listen:

```
list  -> '[' nat tail ']'
tail  -> ',' nat tail | ε
```

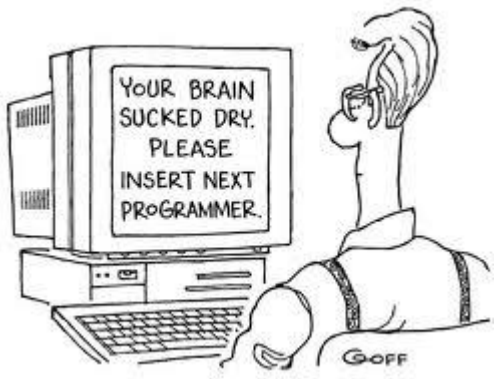
Parser für Listen

```
plist1    :: Parser [Int]
plist1    =  do
               string "["
               n <- nat
               ns <- many (
                  do
                     string ","
                     nat
                  )
               string "]"
               return (n:ns)
```

Parser für Listen

Ergebnis?

```
parse plist1 "[1,2,3]"
parse plist1 "[1,2,]"
parse plist1 "[1,2,3"
parse plist1 "[1, 2, 3]"
```



Parser für Listen

Quiz

Was fehlte dem Parser?

Whitespace

Behandlung von Whitespaces.

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

- Leeres Tupel = Dummy Wert.

Whitespace

Behandlung von Whitespaces.

```
token       :: Parser a -> Parser a
token p      = do
                  space
                  v <- p
                  space
                  return v
```

Whitespace

Daraus ergeben sich...

```
natural  :: Parser Int
natural  = token nat

symbol      :: String → Parser String
symbol xs   = token (string xs)
```

Parser für Listen - Versuch 2

```
plist2    :: Parser [Int]
plist2    =  do
               symbol "["
               n <- natural
               ns <- many (
                  do
                  symbol ","
                  natural
               )
               symbol "]"
               return (n:ns)
```

Arithmetische Ausdrücke

Parser für arithmetische Ausdrücke von natürlichen Zahlen mit den Operatoren + und *.

Regeln:

- Rechtsassoziativ
- Punkt vor Strich

Arithmetische Ausdrücke

Die kontextfreie Grammatik des Parsers:

```
expr   -> term '+' expr | term

term   -> factor '*' term | factor

factor -> '(' expr ')' | natural
```

Arithmetische Ausdrücke

Oder faktorisiert & effizienter:

```
expr   -> term ('+' expr | ε )

term   -> factor ('*' term | ε )

factor -> '(' expr ')' | natural
```

Arithmetische Ausdrücke

Die Grammatik übertragen in Haskell Code:

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

expr   -> term ('+' expr | ε )
```

Arithmetische Ausdrücke

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

```
term   -> factor ('*' term | ε )
```

Arithmetische Ausdrücke

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

```
factor -> '(' expr ')' | natural
```

Arithmetische Ausdrücke

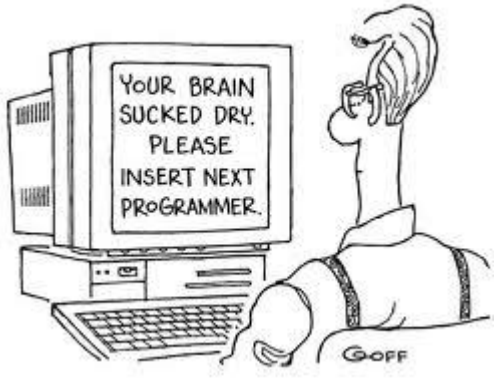
Die Auswertung:

```
eval :: String -> Int
eval xs =  case parse expr xs of
        Just (n,[])  -> n
        Just (_,out) -> error ("unused input
" ++ out)
        Nothing      -> error "invalid input"
```

Arithmetische Ausdrücke

Ergebnis?

```
eval "2*3+4"
eval "2*(3+4)"
eval "2*3-4"
eval "[3+4]+3"
```



Fragen?

Vielen Dank für eure Aufmerksamkeit :-)