# Haskell Workshop - Beispiel Parser-Kombinatoren

Carsten König

Juni 2021

## Parser - Kombinatoren

die Idee von einem Parser ist folgende:

- der Parser bekommt einen String als Input
- der Parser kann jetzt den Anfang des String untersuchen so weit er will, dabei *konsumiert* er diesen
- als Ergebnis liefert der Parser einen erkannten Wert (kann irgendein Typ sein) und den Rest des Strings, den er nicht *konsumiert* hat

```haskell
newtype Parser a
  = Parser
  { runParser :: String -> Maybe (a, String)
  }


parse :: Parser a -> String -> Maybe a
parse pa = fmap fst . runParser pa
```

---

### Übung

Versuche die Funktionen in `Parser.hs` im `step-1` Branch so zu implementieren, dass die Tests alle grün werden

#### Lösung

```haskell
succeed :: a -> Parser a
succeed x = Parser $ \s -> Just (x, s)


fail :: Parser a
fail = Parser $ const Nothing


one :: Parser Char
one = Parser $ \case
  (c:rest) -> Just (c, rest)
  _        -> Nothing


digit :: Parser Char
digit = Parser $ \case
  (c:rest) | isDigit c -> Just (c, rest)
  _                    -> Nothing
```

---

`one` und `digit` kann man ganz gut so zusammenfassen:

```haskell
one :: Parser Char
one = char (const True)


digit :: Parser Char
digit = char isDigit


char :: (Char -> Bool) -> Parser Char
char praed = Parser $ \case
  (c:s) | praed c -> Just (c, s)
  _               -> Nothing
```

# Funktoren

Bisher liefern alle unsere Parser immer nur `Char` Werte. Was ist aber, wenn wir aus dem `digit` eine Zahl machen wollen?

Müssen wir dann einen neuen Parser schreiben? Eigentlich wissen wir doch, dass wir mit `read . (\c -> [c])` aus einem `Char` ein `Int` machen können.

Die Standardantwort in Haskell ist dafür, dass ein `Parser` ein sogenannter **Funktor** ist.

---

## Intuition:

- *Kontainer* mit einem oder mehr Werte
- *Berechnung* die einen Wert produziert
- `fmap` bringt eine Funktion, die Werte transformiert in den Funktor

## Typklasse

```haskell
:i Functor
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a
    {- MINIMAL fmap #-}
```

`<$>` ist Operator für `fmap`

## Laws

- `fmap id = id`
- `fmap (g . h) = (fmap g) . (fmap h)`

## Beispiele

- Listen
- `Maybe`
- `IO`
- `Either a`
- `((->) r)`
- `((,) a)`

```
> fmap succ [1..5]
[2,3,4,5,6]
```

```
> fmap succ (Just 5)
Just 6

> fmap succ Nothing
Nothing

> fmap (++ "!") getLine
Hello
"Hello!"

> fmap succ (Right 5)
Right 6

> fmap succ (Left "argh")
Left "argh"

> fmap succ (*5) 2
11

> fmap succ ("Hallo", 41)
("Hallo",42)
```

## Extension

mit `{- LANGUAGE DeriveFunctor #-}` kann fast jeder ADT zu einem Funktor gemacht werden (mittels `deriving Functor`)

## Übungen

- Functor Instanz für `data Pair a = Pair a a`
- Functor Instanz für `data ITree a = Leaf (Int -> a) | Node [ITree a]`
- Gesucht Typ mit Kind `* -> *`, der kein Functor ist

## Functor sind sind *komposierbar*

### Komposition

```haskell
newtype Compose f g a = Compose (f (g a))
  deriving Show


instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose $ fmap (fmap f) x


type MaybeList a = Compose [] Maybe a

example :: MaybeList String
example = fmap show $ Compose [Just 5, Nothing, Just 4]
```

### Coprodukt

```haskell
newtype Coproduct f g a = Coproduct (Either (f a) (g a))
  deriving Show


instance (Functor f, Functor g) => Functor (Coproduct f g) where
```

```
  fmap f (Coproduct (Left x)) = Coproduct (Left $ fmap f x)
  fmap f (Coproduct (Right y)) = Coproduct (Right $ fmap f y)


type MaybeOrList a = Coproduct Maybe [] a

exampleL :: MaybeOrList String
exampleL = fmap show $ Coproduct (Left $ Just 5)

exampleR :: MaybeOrList String
exampleR = fmap show $ Coproduct (Right $ [7])
```

### *exotische* Funktoren

```
newtype Ident a = Ident { runId :: a }

instance Functor Ident where
  fmap f (Ident a) = Ident (f a)


newtype Const b a = Const { runConst :: b }

instance Functor (Const b) where
  fmap f (Const b) = Const b
```

### Einschub: `fix`

```
fix :: (a -> a) -> a
```

Beispiele:

```
fix (1:) == [1,1,1,..]
fix (\c x -> if x <= 1 then 1 else x * c (x-1)) 5 == 120
```

Für Typen:

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

### heftiges Beispiel Recursion Schemes

```
{- LANGUAGE DeriveFunctor #-}

module RecSchemes where


data ExprF a
  = ValF Int
  | AddF a a
  | MulF a a
  deriving (Show, Eq, Functor)


type Expr = Fix ExprF


data Fix f = Fix { unFix :: f (Fix f) }


val :: Int -> Expr
val n = Fix $ ValF n
```

```haskell
add :: Expr -> Expr -> Expr
add a b = Fix $ AddF a b

mul :: Expr -> Expr -> Expr
mul a b = Fix $ MulF a b


type Algebra f a = f a -> a


evalAlg :: Algebra ExprF Int
evalAlg (ValF n) = n
evalAlg (AddF a b) = a + b
evalAlg (MulF a b) = a * b


eval :: Expr -> Int
eval = cata evalAlg


cata :: Functor f => Algebra f a -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unFix


showAlg :: Algebra ExprF String
showAlg (ValF n) = show n
showAlg (AddF a b) = "(" ++ a ++ " + " ++ b ++ ")"
showAlg (MulF a b) = a ++ " * " ++ b


showExpr :: Expr -> String
showExpr = cata showAlg
```

---

## Übung:

die Funktor-Instanz im `step-2` Branch implementieren, so dass die Tests grün sind.

### Lösung

```haskell
instance Functor Parser where
  fmap f p = Parser $ \s ->
    case runParser p s of
      Just (x, rest) -> Just (f x, rest)
      Nothing        -> Nothing
```

für später brauchen wir noch einen `try` Parser-Combinator:

falls `p` für eine Eingabe fehlschlägt, soll `try p` erfolgreich sein, ohne Input zu *konsumieren*, aber als Ergebnis `Nothing` liefern.

falls `p` erfolgreich mit Ergebnis `x` ist, soll `try p` die gleiche Eingabe *konsumieren* und Ergebnis `Just x` liefern:

```haskell
try :: Parser a -> Parser (Maybe a)
try pa = Parser $ \s ->
  case runParser pa s of
    Nothing      -> Just (Nothing, s)
    Just (a, s') -> Just (Just a, s')
```

ein mit `try` kombinierter Parser kann also nicht schief laufen.

> **Hinweis:** ist in `step-3` enthalten

# Mehrere Parser miteinander kombinieren

## Applicative

## Intuition

was, wenn die Funktion, die wir anwenden wollen, selbst im *Kontext verpackt ist*?

mit `Applicative` können *verpackte* Funktionen auf *verpackte* Werte anwenden

## Typklasse

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

## Laws

- Identity: `pure id <*> v = v`
- Homomorphism: `pure f <*> pure x = pure (f x)`
- Interchange: `u <*> pure y = pure ($ y) <*> u` (die Reihenfolge der Auswertung ist egal)
- Composition: `u <*> (v <*> w) = pure (.) <*> u <*> v <*> w` (ein wenig wie Assoziativität)

mit Funktor:

- `g <$> x = pure g <*> x`

## Beispiele

### Maybe

```
> Just (+) <*> Just 5 <*> Just 6
Just 11

> Just (+) <*> Nothing <*> Just 6
Nothing

> Nothing <*> Just 5
Nothing
```

### Listen

Es gibt zwei Möglichkeiten Listen zu einer Instanz von `Applicative` zu machen:

- Funktionen-Liste und Werte-Liste paarzweise anwenden
- jede Funktion in der Funktionen-Liste mit jedem Wert in der Werte-Liste applizieren

---

### Übung

Newtypes für die beiden Möglichkeiten und Applicative implementieren

### Lösung

```
newtype ZipList a = ZipList [a]
  deriving (Show, Functor)

instance Applicative ZipList where
  pure = ZipList . repeat
  ZipList fs <*> ZipList xs = ZipList $ zipWith ($) fs xs


newtype CompList a = CompList [a]
  deriving (Show, Functor)

instance Applicative CompList where
  pure a = CompList [a]
  CompList fs <*> CompList xs =
    CompList $ [ f x | f <- fs, x <- xs ]
```

---

### Paare

Wie könnte eine Instanz für `(,) a` aussehen?

- Für `pure x :: xludwigsstadt mietpreis pro qm -> (a,x)` brauchen wir einen Wert für `a0 :: a`
- Für `(a1,f) <*> (a2,x) = (a1 ? a2, f x)` müssen wir uns etwas für `? :: a -> a -> a` ausdenken
  - `pure id <*> (a,x) = (a0, id) <*> (a0 ? a x) =(law)= (a, x)` - d.h. `a0 ? a = a`
  - analog wegen interchange für `a ? a0 = a`
  - Composition schließlich heißt `a1 ? (a2 ? a3) = (a1 ? a2) ? a3`
  - eine derartige Operation heißt **Monoid** und dafür gibt es bereits eine Typklasse

```
newtype Paar a b = Paar (a, b)
  deriving (Show, Functor)

instance Monoid a => Applicative (Paar a) where
  pure a = Paar (mempty, a)
  Paar (a,f) <*> Paar (b,x) = Paar (a `mappend` b, f x)
```

---

### Übung

Implemntiere `sequenceAL :: Applicative f => [f a] -> f [a]`

### Lösung

```
sequenceAL :: Applicative f => [f a] -> f [a]
sequenceAL = foldr (liftA2 (:)) (pure [])
```

---

## Übung Applikative für Parser

```
git checkout step-3
```

### Lösung

```
instance Applicative Parser where
  pure = succeed
  pf <*> pa = Parser $ \s -> do
    (f, s') <- runParser pf s
```

```
    (x, s'') <- runParser pa s'
    return (f x, s'')
```

---

Wir können verschiedene Parser verknüpfen, aber was ist, wenn wir eine Alterntive brauchen?

`many` diskutieren

Glücklicherweise gibt es auch hierfür eine Typklasse: Alternative

---

### Übung

Implementiere Alterntive

```
git checkout step-4
```

#### Lösung

```
instance Alternative Parser where
  empty = Parser.fail
  p1 <|> p2 = Parser $ \s ->
    case runParser p1 s of
      ok@(Just _) -> ok
      Nothing     -> runParser p2 s
```

damit bekommen wir `many` direkt geschenkt

und `many1` ist nicht viel schwieriger

```
many1 :: Parser a -> Parser [a]
many1 = some


oneOf :: [Parser a] -> Parser a
oneOf = foldr (<|>) empty
```

---

## Monoid / Monad

```
git checkout step-5
```

```
instance Monoid a => Monoid (Parser a) where
  mempty          = succeed mempty
  p1 `mappend` p2 = mappend <$> p1 <*> p2


instance Monad Parser where
  return     = pure
  pa >>= fpb = Parser $ \s -> do
    (a, s') <- runParser pa s
    runParser (fpb a) s'
```

---

## der Rest

```
git checkout step-6
```

```haskell
chainl1 :: forall a . Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 pa pop = pa >>= more
  where
    more a =
      do
        op <- pop
        a' <- pa
        more (a `op` a')
      <|> succeed a


chainl :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl pa pop va = chainl1 pa pop <|> succeed va


between :: Parser l -> Parser r -> Parser a -> Parser a
between pl pr pa = pl *> pa <* pr


whitespace :: Parser ()
whitespace = return () <* many (char (`elem` " \n\t\r"))
```

### Challange:

Versuche `chainl1` nur mit `Applictive` zu implementieren

---

## Formel-Parser

```
git checkout step-6
```

### Lösung

```haskell
parse :: (Fractional a, Num a, Read a) => String -> Maybe (Formel a)
parse = P.parse formelP


formelP :: (Fractional a, Num a, Read a) => P.Parser (Formel a)
formelP = strichP


strichP :: (Fractional a, Num a, Read a) => Read a => P.Parser (Formel a)
strichP = P.chainl1 punktP opsP
  where opsP = P.oneOf $ map (uncurry opP) [ ('+',(+)), ('-',(-)) ]


punktP :: (Fractional a, Num a, Read a) => P.Parser (Formel a)
punktP = P.chainl1 konstP opsP
  where opsP = P.oneOf $ map (uncurry opP) [ ('*', (*)), ('/', (/)) ]


konstP :: (Fractional a, Num a, Read a) => P.Parser (Formel a)
konstP = P.oneOf
  [ P.between (chWs '(') (chWs ')') formelP
  , Konstante <$> numberP <* P.whitespace ]
```

```haskell
opP :: Char -> a -> P.Parser a
opP c a = chWs c *> pure a


chWs :: Char -> P.Parser ()
chWs c = P.char (== c) *> P.whitespace


numberP :: (Fractional a, Read a) => P.Parser a
numberP =
  read <$> mconcat [vorzP, ziffernP, kommaP]
  where
    vorzP    = maybe "" return <$> P.try (P.char (== '-'))
    ziffernP = P.many1 P.digit
    kommaP   = fromMaybe "" <$> P.try (mconcat [pure <$> P.char (== '.'), P.many1 P.digit])
```

---

## Scotty App

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE OverloadedStrings #-}

module Main where

import           Control.Monad.IO.Class (liftIO)
import           Data.Monoid ((<>))
import qualified Data.Text as T
import           Formel
import           Lucid
import qualified Lucid.Html5 as H
import           Network.Wai (Middleware)
import           Network.Wai.Middleware.Static (staticPolicy, addBase, noDots)
import           System.Environment (lookupEnv)
import           Web.Scotty


main :: IO ()
main = do
  port <- maybe 80 read <$> lookupEnv "PORT"

  scotty port $ do
    -- serve static files form "./static" folder
    middleware serveStaticFiles
    -- post formulas as string in json-body that will get evaluated and sent back as a json-number
    -- try it: curl -H "Content-Type: application/json" -X POST -d '"3 + 3 * 5 / 30 - 22.22"' http://lo
    post "/eval" $ do
      formel <- jsonData
      liftIO $ print formel
      maybe (raise "konnte Formel nicht parsen") (json . eval) $ (parse formel :: Maybe (Formel Double)
    get "" $ do
      html $ renderText (page Nothing)
    post "" $ do
      formel <- param "formel"
      let result = eval <$> parse formel
      html $ renderText (page $ (formel,) <$>  result)


serveStaticFiles :: Middleware
```

```
serveStaticFiles = staticPolicy (noDots <> addBase "static")


page :: Maybe (String, Double) -> Html ()
page result = H.doctypehtml_ $ do
  H.head_ $ do
    H.meta_ [ H.charset_ "UTF-8" ]
    H.meta_ [ H.name_ "viewport", H.content_ "width=device-width, initial-scale=1, shrink-to-fit=no" ]
    H.link_ [ H.rel_ "stylesheet", H.href_ "bootstrap.css" ]

    H.title_  "Rechner"

  H.body_ $ do
    H.div_ [ H.class_ "content" ] $ do
      caption
      inputForm

    script "jquery.js"
    script "pooper.js"
    script "bootstrap.js"

  where
    script fileName =
      H.script_ [ H.src_ fileName ] ("" :: String)

    caption =
      H.div_ [ H.class_ "jumbotron" ] $
        H.h1_ "Rechner..."

    inputForm =
      H.form_ [ H.method_ "post"
              , H.class_ "form-inline"
              ] $ do
        H.div_ [ H.class_ "form-group" ] $
          H.input_ [ H.type_ "text"
                   , H.class_ "form-control"
                   , case result of
                       Just (formel,_) -> H.value_ $ T.pack formel
                       Nothing         -> H.placeholder_ "Formel..."
                   , H.autofocus_
                   , H.name_ "formel"
                   ]
        showResult
        H.button_ [ H.type_ "submit"
                  , H.class_ "btn btn-primary"
                  ] "calc"

    showResult =
      case result of
        Nothing  -> return ()
        Just (_, erg) -> H.div_ [ H.class_ "form-group" ] $
          H.input_ [ H.type_ "text"
                   , H.readonly_ ""
                   , H.class_ "form-control-plaintext"
                   , H.value_ $ T.pack $ " = " ++ show erg
                   ]
```