

Ein wichtiger (und nützlicher) Spezialfall

- Häufig tritt folgender Spezialfall auf:

```
do  $x_1 \leftarrow \text{prs}_1$   
    $x_2 \leftarrow \text{prs}_2$   
   ...  
    $x_n \leftarrow \text{prs}_n$   
return (f  $x_1 x_2 \dots x_n$ )
```

wobei f eine normale Funktion ist und keines der prs_j von irgendeinem x_i abhängt (und wobei durchaus auch einige x_i ganz weggelassen sein können).

- Dann ist die Vergabe von Namen für die Ergebnisse der prs_i eigentlich überflüssig, nur die Reihenfolge ist wesentlich.
- Für diesen Fall kann dann folgende vereinfachte Form verwendet werden:

```
f <$> prs1 <*> prs2 <*> ... <*> prsn
```

Ein wichtiger (und nützlicher) Spezialfall

- Ermöglicht wird dies durch zusätzlich zur Verfügung stehende Kombinatoren:

```
infixl 4 <$>, <*>
```

```
(<$>) :: (a → b) → Parser a → Parser b
```

```
(<*>) :: Parser (a → b) → Parser a → Parser b
```

(sowie Varianten `<$`, `<*`, `*>`, siehe Dokumentation von [Control.Applicative](#)),
welche durch folgende minimale Definition realisiert sind:

```
instance Functor Parser where  
  fmap = mapP
```

```
instance Applicative Parser where  
  pure   = yield  
  (<*>) = liftP ($)
```

(restliche Definitionen ergeben sich daraus).

Ein wichtiger (und nützlicher) Spezialfall

- Nun erschließt sich (hoffentlich) auch die ursprünglich angegebene Lösung für das Parsen arithmetischer Ausdrücke:

```
expr  = ( Add <$> term <*> char '+' <*> expr ) ||| term  
  
term  = ( Mul <$> factor <*> char '*' <*> term ) ||| factor  
  
factor = ( Lit <$> nat ) ||| ( char '(' *> expr <*> char ')' )
```

- Denn, zum Beispiel, die erste Zeile steht für:

```
expr = ( pure (\t _ e → Add t e) <*> term <*> char '+' <*> expr )  
      ||| term
```

und somit für:

```
expr = do t ← term  
        _ ← char '+'  
        e ← expr  
        return (Add t e)  
      ||| term
```

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Zur Erinnerung:

```
> parse (liftP (,) lower digit) "a1"  
('a', '1')
```

bzw.:

```
> parse ((,) <$> lower <*> digit) "a1"  
('a', '1')
```

- Auch noch erwartbar:

```
> parse (liftP (,) (many1 lower) (many1 digit)) "abc123"  
("abc", "123")
```

- Aber, problematisch:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"  
Program error: invalid input
```

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Ursache für:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"  
Program error: invalid input
```

muss irgendwo liegen in:

```
liftP :: (a → b → c) → Parser a → Parser b → Parser c  
liftP f p q = p ++> \x → q ++> \y → yield (f x y)
```

```
many1 :: Parser a → Parser [a]  
many1 p = liftP (:) p (many1 p ||| yield [ ])
```

```
(|||) :: Parser a → Parser a → Parser a  
p ||| q = \inp → case p inp of  
    Just (x, rest) → Just (x, rest)  
    Nothing       → q inp
```

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Tatsächliche Ursache:

```
many1 :: Parser a → Parser [a]  
many1 p = liftP (:) p (many1 p ||| yield [ ])
```

und

```
(|||) :: Parser a → Parser a → Parser a  
p ||| q = \inp → case p inp of  
    Just (x, rest) → Just (x, rest)  
    Nothing       → q inp
```

implizieren „greedy matching“.

- Das heißt, das erste `many1 lower` in

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"
```

„verbraucht“ die gesamte Eingabe "abcdef".

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Vielleicht sollten wir also die Reihenfolge in `many1` vertauschen?

```
many1 :: Parser a → Parser [a]
many1 p = liftP (:) p (yield [ ] ||| many1 p)
```

- Nicht wirklich besser:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"
Program error: unused input: cdef
```

- Das eigentliche Problem ist die willkürliche Bevorzugung von `p` in:

```
(| | |) :: Parser a → Parser a → Parser a
p | | | q = \inp → case p inp of
    Just (x, rest) → Just (x, rest)
    Nothing       → q inp
```

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Um p und q in

```
(| | |) :: Parser a → Parser a → Parser a  
p | | | q = \inp → case p inp of  
    Just (x, rest) → Just (x, rest)  
    Nothing       → q inp
```

gleich zu behandeln, Abkehr von

```
type Parser a = String → Maybe (a, String)
```

nötig.

- Idee:

```
type Parser a = String → [ (a, String) ]
```

- Dann:

```
(| | |) :: Parser a → Parser a → Parser a  
p | | | q = \inp → p inp ++ q inp
```


Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Natürlich auch Anpassungen anderer Funktionen aus `ParserCore.hs` nötig:

```
type Parser a      parse    item    yield    failure  (| |)  (++)>
```

- Einige Änderungen sehr leicht:

```
failure :: Parser a  
failure = \inp → Nothing
```



```
failure :: Parser a  
failure = \inp → [ ]
```

```
yield :: a → Parser a  
yield x = \inp → Just (x, inp)
```



```
yield :: a → Parser a  
yield x = \inp → [ (x, inp) ]
```

```
item :: Parser Char  
item = \inp → case inp of  
    ""      → Nothing  
    x : xs  → Just (x, xs)
```



```
item :: Parser Char  
item = \inp → case inp of  
    ""      → [ ]  
    x : xs  → [ (x, xs) ]
```

- Andere Anpassungen erfordern etwas mehr Einsicht ...

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

```
(++>) :: Parser a → (a → Parser b) → Parser b  
p ++> f = \inp → case p inp of  
    Nothing      → Nothing  
    Just (x, rest) → f x rest
```



```
(++>) :: Parser a → (a → Parser b) → Parser b  
p ++> f = \inp → concatMap (\(x, rest) → f x rest) (p inp)
```

```
parse :: Parser a → String → a  
parse p inp = case p inp of  
    Nothing      → error "invalid input"  
    Just (x, "")  → x  
    Just (_, rest) → error ("unused input: " ++ rest)
```



```
parse :: Parser a → String → [a]  
parse p inp = [ x | (x, rest) ← p inp, rest == "" ]
```

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Andere Funktionen, in `Parser.hs`:

```
(+++)  
digit  lower  ...  sat  liftP  mapP  
many   many1  char
```

brauchen **nicht** angepasst zu werden!

- Nach Ersetzung von `ParserCore.hs` durch `LParserCore.hs`, jetzt:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"  
[("abcde","f"), ("abcd","ef"), ("abc","def"), ("ab","cdef"), ("a","bcdef")]
```

bzw. (bei `yield [] ||| many1 p` statt `many1 p ||| yield []` in `many1`):

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"  
[("a","bcdef"), ("ab","cdef"), ("abc","def"), ("abcd","ef"), ("abcde","f")]
```

- Alternativen `ParserCore.hs`:

```
type Parser a = String → Maybe (a, String)
parse      ...      (| |)      (++)>
```

oder `LParserCore.hs`:

```
type Parser a = String → [ (a, String) ]
parse      ...      (| |)      (++)>
```

- `MParserCore.hs` (importiert `ParserCore.hs` oder `LParserCore.hs`), zur Verwendung von do-Notation, `<$>`, `<*>`, ...; und macht `Parser`-Typ abstrakt/opak

- `Parser.hs`:

```
(+++)  
digit  lower  ...  sat  liftP  mapP  
  
many   many1  char
```

- Verwendung von `LParserCore.hs` liefert immer eine Obermenge (oder die gleiche Menge) der Parse-Ergebnisse bei Verwendung von `ParserCore.hs`.
- Bei „deterministischen Grammatiken“, kein Unterschied!

- Alternativen `ParserCore.hs`:

```
type Parser a = String → Maybe (a, String)
parse      ...      (| |)      (++)>
```

oder `LParserCore.hs`:

```
type Parser a = String → [ (a, String) ]
```

- `MParser`
von `do-N`

- `Parser.hs`

`(+++)`

`many`

- Verwend
Menge)

- Bei „det

Dr. Seuss on Parser Monads:



```
type Parser a = String → [(a, String)]
```

A Parser for Things
is a function from Strings
to Lists of Pairs
of Things and Strings!

Art: Seuss; Type: Wadler; Rhyme: Rueter

Haiku

*To recurse through lists,
Simply work on beginnings
Until it's the end*

Tom Murphy

Haiku

*With strongly typed hands,
I recurse guardingly
in comprehensive repose.*

Todd Coram

The Poetry of Errors

*I think there's a case that I missed,
For GHC seems to insist,
That when I run main,
it is all in vain:*

**** Exception: Prelude.head: empty list*

Wouter Swierstra

Mehr in: „Special Poetry and Fiction
Edition of The Monad.Reader“

Betrachtung der verwendeten Hauptabstraktion (das „M-Wort“)

- Wir hatten nun, bzgl. **do-Notation**, im Fall von Ein- und Ausgabe:

```
do cmd1  
  x2 ← cmd2  
  x3 ← cmd3  
  cmd4  
  x5 ← cmd5  
  ...
```

wobei die **cmd_i** jeweils IO-Typen haben und an die **x_i** (sofern explizit vorhanden) jeweils ein Wert des in **cmd_i** gekapselten Typs gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das Ergebnis der Ausführung von **cmd_i**.

- Im Fall von Parsern (in sowohl der **ParserCore**- als auch der **LParserCore**-Variante):

```
do prs1  
  x2 ← prs2  
  x3 ← prs3  
  prs4  
  x5 ← prs5  
  ...
```

wobei die **prs_i** jeweils einen Typ der Form **Parser t_i** haben und an die **x_i** (sofern explizit vorhanden) dann jeweils ein Wert des Typs **t_i** gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das durch **prs_i** gelieferte Ergebnis.

- Das ruft nach Verallgemeinerung!

Betrachtung der verwendeten Hauptabstraktion (das „M-Wort“)

- Tatsächlich gibt es eine ganze Klasse auf diesem Interface basierender „domänenspezifischer Sprachen“.
- Beispiele, neben 1., **IO** (mit `getChar`, `putChar`, ...) und 2./3., **Parser** (mit `item`, `|||`, ...):

4.

```
do exp1  
  x2 ← exp2  
  x3 ← exp3  
  exp4  
  x5 ← exp5  
  ...
```

wobei die `expi` jeweils einen Typ der Form `Maybe ti` haben ...

und wobei der gesamte do-Block genau dann zu `Nothing` ausgewertet wenn es irgendeines der `expi` tut.

5.

```
do exp1  
  x2 ← exp2  
  x3 ← exp3  
  exp4  
  x5 ← exp5  
  ...
```

wobei die `expi` jeweils einen Typ der Form `Either String ti` haben ...

und wobei der gesamte do-Block genau dann zu einem `Left`-Wert (mit String-„Fehlermeldung“) ausgewertet wenn es irgendeines der `expi` tut.

Betrachtung der verwendeten Hauptabstraktion (das „M-Wort“)

- Anwendungsbeispiel: zur Erinnerung, wir hatten ...

```
eval :: Expr → Maybe Int
eval (Lit n)      = Just n
eval (Add e1 e2) = case eval e1 of
                        Nothing → Nothing
                        Just r1 → case eval e2 of
                                    Nothing → Nothing
                                    Just r2 → Just (r1 + r2)
...

```

... ersetzt durch:

```
eval (Add e1 e2) = eval e1 `andThen` \r1 →
                      eval e2 `andThen` \r2 → Just (r1 + r2)

```

- Eigentlich noch besser:

```
eval (Add e1 e2) = do r1 ← eval e1
                      r2 ← eval e2
                      return (r1 + r2)

```

Betrachtung der verwendeten Hauptabstraktion (das „M-Wort“)

6.

```
do ndt1  
  x2 ← ndt2  
  x3 ← ndt3  
  ndt4  
  x5 ← ndt5  
  ...
```

wobei die ndt_i jeweils einen Typ der Form $[t_i]$ haben ...

und wobei der gesamte do-Block zu einer Liste aller kombinatorischen Möglichkeiten auswertet.

7.

```
do cmd1  
  x2 ← cmd2  
  x3 ← cmd3  
  cmd4  
  x5 ← cmd5  
  ...
```

wobei die cmd_i jeweils einen Typ der Form $\text{State } s \ t_i$ haben (für einen per do-Block festen Typ s) ...

und wobei der gesamte do-Block eine sequentielle Abarbeitung unter Weitergabe eines veränderlichen Zustands beschreibt.

weitere „Sprachkonstrukte“ in diesem Fall:

```
put :: s → State s ()  
get :: State s s
```

Betrachtung der verwendeten Hauptabstraktion (das „M-Wort“)

8.

```
do dsb1  
  x2 ← dsb2  
  x3 ← dsb3  
  dsb4  
  x5 ← dsb5  
  ...
```

wobei die exp_i jeweils einen Typ der Form $\text{Gen } t_i$ haben (definiert in [QuickCheck](#)) ...

und wobei der gesamte do-Block eine (modular zusammengesetzte) Zufallsverteilung für Werte eines bestimmten Typs beschreibt.

weitere „Sprachkonstrukte“ in diesem Fall:

```
arbitrary  :: Arbitrary a ⇒ Gen a  
elements  :: [a] → Gen a  
oneof      :: [Gen a] → Gen a  
frequency :: [(Int, Gen a)] → Gen a  
suchThat  :: Gen a → (a → Bool) → Gen a  
listOf     :: Gen a → Gen [a]  
vectorOf  :: Int → Gen a → Gen [a]  
...
```

9., 10., 11. ...