## Heutiges Themea

*Parsing*: Lexikalische und syntaktische Analyse

- Kombinatorparsing
- Monadisches Parsing



## Lexikalische und syntaktische Analyse

- ...im folgenden kurz *Parsing*

...eine typische Demonstrationsanwendung funktionaler Programmierung mit langer Tradition.

Eine frühe Arbeit ist etwa...

- W. Burge. *Recursive Programming Techniques*, Addison-Wesley, 1975.



## Parsing – Implementierungsvarianten

Zwei Spielarten...

- *Kombinator-Parsing*
  $\leadsto$ *recursive descent* Parsing
  - Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- *Monadisches Parsing*
  - Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.



## Literaturhinweis

Die folgende Darstellung orientiert sich i.w. an...

- Kapitel 17
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 1(1), 1993.

## Parsing informell

Das Grundproblem...

- Lies eine Folge von Objekten vom Typ `a` und
- extrahiere daraus ein(e Liste von) Objekt(en) vom Typ `b`.



## Beispiel: Parsen von Ausdrücken

Betrachte...

- Ausdrücke

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Mul (Op Add (Lit 2) (Lit 3)) (Lit 3)
                        entspricht  ((2+3)*3)
```

Zu lösende (Parsing-) Aufgabe...

- Einlesen von Ausdrücken der Form `((2+3)*3)` und Ausgeben des zugehörigen Ausdrucks vom Typ `expr`



## Vorüberlegungen 1(2)

Welchen Typ sollte eine Parsing-Funktion haben?

```
type BSParse1 a b = [a] -> b

-- Parser  Eingabe      Erwartete Ausgabe
bracket    "(xyz"  -->  '('
number     "234"   -->  2 oder 23 oder 234 ?
bracket    "234"   -->  kein Ergebnis, Fehlschlag?
```

Zu beantwortende Fragen hierbei: Wie soll sich der Parser verhalten im Falle...

- ...mehrerer Ergebnisse
- ...eines Fehlschlags?



## Vorüberlegungen 2(2)

```
type BSParse2 a b = [a] -> [b]

-- Parser  Eingabe      Erwartete Ausgabe
bracket    "(xyz"  -->  ['(']
number     "234"   -->  [2, 23, 234]
bracket    "234"   -->  []
```

Zu beantwortende Frage hierbei:

- Was passiert mit der restlichen Eingabe?

## Typ des Parsers 1(2)

Ergebnis der Vorüberlegungen...

```
type Parse a b = [a] -> [(b,[a])]

-- Parser  Eingabe      Erwartete Ausgabe

bracket    "(xyz"  -->  [('(', "xyz")]
number     "234"   -->  [(2,"34"), (23,"4"), (234,"")]
bracket    "234"   -->  []
```

*Bemerkung*:

- Ausgabe mehrerer Analyseergebnisse erlaubt die Analyse auch mehrdeutiger Grammatiken
  $\leadsto$ *list of successes* technique



## Typ des Parsers 2(2)

*Vereinbarung*:

- *Ausgabe der leeren Liste* ...repräsentiert Fehlschlag der Analyse
- *Ausgabe einer nichtleeren Liste* ...repräsentiert Erfolg der Analyse; jedes Element der Liste ist dann ein Paar, dessen erste Komponente das erkannte Objekt (token) ist und dessen zweite Komponente die noch nicht gelesene Eingabe ist



## Grundlegende/Basis-Parser 1(3)

- Primitive, eingabeunabhängige Parser

```
-- Der nie erfolgreiche Parser
none :: Parse a b
none inp = []

-- Der stets erfolgreiche Parser
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

*Bemerkung*: Der `succeed`-Parser konsumiert seine Eingabe nicht. In BNF-Notation entspricht er dem $\varepsilon$-Symbol für das leere Wort.



## Grundlegende/Basis-Parser 2(3)

- Erkennung einzelner Objekte (token)...

```
token :: Eq a => a -> Parse a a
token t (x:xs)
  | t == x     =  [(t,xs)]
  | otherwise  =  []
token t []     =  []
```

- Erkennung einzelner Objekte mit einer bestimmten qualifizierenden Eigenschaft...

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x        =  [(x,xs)]
  | otherwise  =  []
spot p []      =  []
```

## Grundlegende/Basis-Parser 3(3)

*Anwendung*:

```
bracket = token '('
dig     = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

*Beachte*: ...`token` mittels `spot` ausdrückbar

```
token t = spot (== t)
```



## Zusammengesetzte/Kombinierte Parser 1(4)

...liefern komplexe(re) Parser

$\leadsto$ *Combinator Parsing*

- Alternativen

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

*Beispiel*:

```
(bracket 'alt' dig) "234"  -->  [] ++ [(2,"34")]
```



## Zusammengesetzte/Kombinierte Parser 2(4)

- Sequenzen/Hintereinanderausführung

```
infixr 5 >*>
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                           (z,rem2) <- p2 rem1 ]
```



## Zusammengesetzte/Kombinierte Parser 3(4)

*Beispiel*:

```
wegen  number "24(" --> [(2,"4("), (24,"(")]  zunaechst

(number >*> bracket) "24("
   -->  [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],
                        (z,rem2) <- bracket rem1 ]
   -->  [((2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
        [((24,z),rem2) | (z,rem2) <- bracket "(" ]

   -->  [] ++ [((24,z),rem2) | (z,rem2) <- bracket "(" ]

wegen  bracket "("  -->  [('(',"")]  somit insgesamt

   -->  [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
   -->  [ ((24,'('), "" ]
```

## Zusammengesetzte/Kombinierte Parser 4(4)

- Transformation/Modifikation

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

*Beispiel*:

```
(digList 'build' digsToNum) "21a3"
   --> [ (digsToNum x,rem) | (x,rem) <- digList "21a3" ]
   --> [ (digsToNum x,rem) | (x,rem) <-
                             [("2","1a3"),("21","a3")]]
   --> [ (digsToNum "2", "1a3"), (digsToNum "21", "a3") ]
   --> [ (2,"1a3"), (21,"a3") ]
```



## Bsp.: Parsen einer Liste von Objekten

...gegeben einen Parser zum Erkennen eines einzelnen Objektes

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```



## Zusammenfassung und Zwischenfazit

...über das Kombinieren von Parsern (*parser combinators*)

- Parser im obigen Stil sind strukturell der Form von BNF-Grammatiken ähnlich. Die Operatoren der BNF-Grammatik haben auf Parserseite eine direkte Entsprechung in Form einer Funktion höherer Ordnung.
- Diese Funktionen höherer Ordnung *kombinieren* einfache(re) Parser zu komplexe(re)n neuen Parsern.
- Man spricht daher von *kombinierenden Formen* (engl. *combining forms*) oder kurz von *Kombinatoren* (engl. *combinators*) (vgl. dazu z.B. Graham Hutton. *Higher-Order Functions for Parsing*.



## Die Parser-Fkt. im Überblick 1(4)

```
-- Sequenzoperator
infixr 5 >*>

-- Parser-Typ
type Parse a b = [a] -> [(b,[a])]

-- Eingabeunabhaengige Parser
none :: Parse a b
none inp = []

succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

## Die Parser-Fkt. im Überblick 2(4)

```
-- Erkennen einzelner Objekte
token :: Eq a => a -> Parse a a
token t = spot (==t)

-- Erkennen einzelner Objekte mit bestimmter Eigenschaft
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x       =  [(x,xs)]
  | otherwise  =  []
spot p []     =  []
```



## Die Parser-Fkt. im Überblick 3(4)

```
-- Alternativen
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp

-- Sequenzen
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp, (z,rem2) <- p2 rem1 ]

-- Transformation/Modifikation
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```



## Die Parser-Fkt. im Überblick 4(4)

```
-- Anwendungsbeispiel
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```



## Anwendung: Zurück zum Ausgangsbeispiel

Wir betrachten Ausdrücke der Form...

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Add (Lit 2) (Lit 3)     entspricht   2+3
```

...wobei weiter die folgende Vereinbarung gelte:

- *Literale* ...67, ∼89, etc., mit ∼ für einstelliges Minus
- *Namen* ...die Kleinbuchstaben von 'a' bis 'z'
- *Anwendungen der binären Operationen* ...$+, *, -, /, \%$, wobei % für die modulo-Operation steht und / die ganzzahlige Division bezeichnet.

## Ein Parser für Ausdrücke 1(3)

```
parser :: Parse Char Expr
parser = litParse 'alt' nameParse 'alt' opExpParse

nameParse :: Parse Char Expr
nameParse = spot isName 'build' Name

isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z')
```



## Ein Parser für Ausdrücke 2(3)

```
opExpParse
  = (token '(' >*>
     parser    >*>
     spot isOp >*>
     parser    >*>
     token ')')
     'build' makeExpr

litParse
  = ((optional (token '~')) >*>
     (neList (spot isDigit))
    'build' (charlistToExpr . uncurry (++))
```



## Ein Parser für Ausdrücke 3(3)

Verwendete Hilfsfunktionen wie...

- `isOp`
- `charlistToExpr`
- ...

bleiben noch zu definieren.



## Top-level Parser

```
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error ``parse unsuccessful''
      _  -> head results
    where
    results = [ found | (found, []) <- p inp ]
```

## Zusammenfassung und Fazit 1(2)

Parser der Form...

```
type Parse a b = [a] -> [(b,[a])]

none :: Parse a b
succeed :: b -> Parse a b
spot :: (a -> Bool) -> Parse a a
alt :: Parse a b -> Parse a b -> Parse a b
>*> :: Parse a b -> Parse a c -> Parse a (b,c)
build :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

...unterstützen besonders die Konstruktion sog. *recursive descent* Parser.



## Zusammenfassung und Fazit 2(2)

Folgende Spracheigenschaften werden besonders ausgenutzt...

- *Funktionen höherer Ordnung* ...`Parse a b` ist von einem Funktionstyp, somit sind alle Parserkombinatoren Funktionale
- *Polymorphie* ...siehe wieder Typ von `Parse a b`, obige Parserkombinatoren sind für andere (Token-) und Zieltypen unmittelbar wiederverwendbar
- *Lazy evaluation* ... *"on demand"*-Erzeugung der möglichen Parsegänge, automatisches backtracking



## Monadisches Parsing

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

*Vereinbarung* auch hier:

- *Ausgabe der leeren Liste* ...repräsentiert Fehlschlag der Analyse
- *Ausgabe einer nichtleeren Liste* ...repräsentiert Erfolg der Analyse; jedes Element der Liste ist dann ein Paar, dessen erste Komponente das erkannte Objekt (token) ist und dessen zweite Komponente die noch nicht gelesene Eingabe ist



## Grundlegende/Basis-Parser

- Erkennung eines einzelnen Zeichens...

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

*Vgl.*: `item` vs. `token`

## Die Parser-Monade

*Erinnerung*: Die Klasse Monade...

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

*Beachte*: `Parser` ist ein Typkonstruktor. Dies ermöglicht...

```
instance Monad Parser where
   -- Der stets erfolgreiche Parser
   return a = Parser (\cs -> [(a,cs)])
   -- Sequenzen/Hintereinanderausfuehrung
   p >>= f  = Parser (\cs -> concat [parse (f a) cs' |
                              (a,cs') <- parse p cs])
```

*Vgl.*: `return` vs. `succeed` und `(>>=)` vs. `infixr`



## Eigenschaften von `return` und `(>>=)`

Wie für Instanzen der Klasse `Monad` gefordert, lässt sich zeigen...

```
             return a >>= f  =  f a
               p >>= return  =  p
p >>= (\a -> (f a >>= g))  =  (p >>= (\a -> f a)) >>= g
```

*Erinnerung*:

- Obige Eigenschaften sind für jede Instanz der Klasse `Monad` gefordert, nicht nur für den Spezialfall der Parser-Monade
  - ...`return` ist Links- und Rechtseins für `(>>=)`
    $\leadsto$ ...erlaubt einige Parser einfacher anzugeben
  - ...`(>>=)` ist assoziativ
    $\leadsto$ ...erlaubt Klammern bei Hinterausführung von Parsern zu unterdrücken



## Typischer Aufbau eines Parsers 1(2)

...mit Hilfe des `(>>=)`-Operators

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

*Intuition*:

- Wende Parser `p1` an und nenne dessen Resultat `a1`
- Wende anschließend Parser `p2` an und nenne dessen Resultat `a2`
- ...
- Wende schließlich Parser `pn` an und nenne dessen Resultat `an`
- Kombiniere abschließend die Teilergebnisse durch Anwendung einer geeigneten Funktion `f`



## Typischer Aufbau eines Parsers 2(2)

Die `do`-Notation erlaubt folgende elegantere Notation...

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Alternativ in einer Zeile notiert...

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an}
```

## Sprech- und Schreibweisen

Ausdrücke der Form

- `ai <- pi` heißen *Generatoren*
  (da sie Werte für die Variablen `ai` erzeugen)

*Bemerkung*:

Ein Generator der Form `ai <- pi` kann durch

- `pi` ersetzt werden, wenn der erzeugte Wert nicht weiter benötigt wird



## Beispiel

```
p :: Parser (Char,Char)
p  = do {c <- item; item; d <- item; return (c,d)}
```

*Informell*: Parser `p`...

- liest drei Zeichen
- verwirft davon das zweite Zeichen und
- gibt das erste und dritte Zeichen als Paar zurück



## Parser-Erweiterungen 1(2)

Monade mit *Null* und `plus`...

```
class Monad m => MonadZero m where
   zero :: m a

class MonadZero m => MonadPlus m where
   (++) :: m a -> m a -> m a
```



## Parser-Erweiterungen 2(2)

```
instance MonadZero Parser where
   zero   = Parser (\cs -> [])

instance MonadPlus Parser where
   p ++ q = (\cs -> parse p cs ++ parse q cs)
```

## Einfache Eigenschaften 1(2)

Es lässt sich zeigen...

```
         zero ++ p  =  p
         p ++ zero  =  p
    p ++ (q ++ r)  =  (p ++ q) ++ r
```

*Bem.*: Obige Eigenschaften sind für jede Monade mit *Null* und `plus` gefordert

*Informell*:

- ...`zero` ist Links- und Rechtseins für (++)
- ...(++) ist assoziativ



## Einfache Eigenschaften 2(2)

Speziell für die Parser-Monade lässt sich zeigen...

```
                 zero >>= f  =  zero
          p >>= const zero  =  zero
            (p ++ q) >>= f  =  (p >>= f) ++ (q >>= f)
 p >>= (\a -> f a ++ g a)  =  (p >>= f) ++ (p >>= g)
```

*Informell*:

- ...`zero` ist Links- und Rechtsnull für (>>=)
- ...(>>=) distribuiert über (++)



## Deterministische Auswahl

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p ++ q) cs of
                             []     -> []
                             (x:xs) -> [x])
```

*Beachte*:

- (+++) zeigt dasselbe Verhalten wie (++), liefert jedoch höchstens ein Resultat ab
- (+++) erfüllt alle der vorher genannten Eigenschaften von (++)



## Weitere Parser

Erkennung...

- einzelner Objekte mit einer bestimmten qualifizierenden Eigenschaft

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

- einzelner Objekte

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

- von Ziffernfolgen, Klein- und Großbuchstaben, etc.
  ...analog zu `char`

*Vgl.*: `sat` und `char` vs. `spot` und `token`

## Rekursions-Kombinatoren 1(3)

Parser können oft rekursiv definiert werden...

```
-- Parsen einer Zeichenreihe
string       :: String -> Parser String
string ""     = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}

-- Parsen wiederholter Anwendungen von Parser p
many  :: Parser a -> Parser [a]
many p = many1 p +++ return []

many1  :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```



## Rekursions-Kombinatoren 2(3)

```
-- wie many mit eingestreuten Anwendungen des Parsers sep
sepby        :: Parser a -> Parser b -> Parser [a]
p 'sepby' sep = (p 'sepby1' sep) +++ return []

sepby1        :: Parser a -> Parser b -> Parser [a]
p 'sepby1' sep = do a  <- p
                    as <- many (do {sep; p})
                    return (a:as)
```



## Rekursions-Kombinatoren 3(3)

```
chainl       :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op a = (p 'chainl1' op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p 'chainl1' op = do {a <- p; rest a}
                 where
                    rest a = (do f <- op
                                 b <- p
                                 rest (f a b))
                             +++ return a
```



## Lexikalische Kombinatoren

Geeignete Kombinatoren erlauben eine lexikalische Analyse (token-Erkennung) zu unterdrücken...

```
-- Parsen einer Zeichenreihe mit Leerstellen und Zeilenumbruechen
space :: Parser String
space  = many (sat isSpace)

-- Parsen eines Tokens mit Hilfe des Parsers p
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}

-- Parsen eines Symboltokens
symb   :: String -> Parser String
symb cs = token (string cs)

-- Anwenden eines Parsers p, entfernen fuehrender Leerstellen
apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p}]
```

## Beispiel: Parsen von Ausdrücken 1(3)

*Grammatik*:

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```



## Beispiel: Parsen von Ausdrücken 2(3)

Parsen und evaluieren von Ausdrücken zu ihrem ganzzahligen Wert...

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr   = term   'chainl1' addop
term   = factor 'chainl1' mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDIgit); return (ord x - ord '0')}

addop  = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop  = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```



## Beispiel: Parsen von Ausdrücken 3(3)

*Beispiel*:

```
apply expr " 1 - 2 * 3 + 4 "
                  -->   [(-1,"")] wie gewuenscht
```



## Weiterführende und vertiefende Literatur 1(3)

Zum Kombinator-Parsing...


- J. Fokker. *Functional Parsers*. In: *Advanced Functional Programming, First International Summer School*, Springer, LNCS 925 (1995), 1-23.

- S. Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6:445-463, 1996.

- P. Koopman, R. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of Implementation of Functional Languages, Springer, LNCS 1595 (1999), 122-138.

## Weiterführende und vertiefende Literatur 2(3)

Zu fehlerkorrigierenden Parsern...


- P. Wadler. *How to Replace Failure with a List of Successes*, in: Functional Programming Languages and Computer Architectures, Springer, LNCS 201 (1985), 113 - 128.
- D. Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer, LNCS 1725 (1999), 111-129.
- D. Swierstra, L. Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer, LNCS 1129 (1996), 184-207.




## Weiterführende und vertiefende Literatur 3(3)

Zu Parser-Bibliotheken...


- Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 2001.
- A. Gill, S. Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995.
`http://www.haskell.org/happy`




## Nächster Vorlesungstermin

- Mi, 14.06.2006, 16 - 18 Uhr, CompLang-Bibliothek