

Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 19.12.2006:
Striktheit und unendliche
Datenstrukturen

Christoph Lüth

WS 06/07



Inhalt

- **Strikte** und **nicht-strikte** Funktionen
 - ... und was wir davon haben.
- Unendliche Datenstrukturen
 - ... und wozu sie nützlich sind.
- Fallbeispiel: Parserkombinatoren

Strikttheit

Def: Funktion f ist **strikt** in einem **Argument** x gdw.

$$x \equiv \perp \implies f(x) \equiv \perp$$

- (+) **strikt** in **beiden** Argumenten.
- (&&) **strikt** im **ersten**, **nicht-strikt** im **zweiten**.

`False && error ~> False`

- `second` **nicht-strikt** im **ersten**:

`second a b = b`

`second undefined 3 ~> 3`

- Zeigen.

Verzögerte Auswertung

- Auswertung: **Reduktion** von Gleichungen
 - **Strategie**: Von **außen** nach **innen**; von **links** nach **rechts**.
- **Effekt**:
 - *call-by-need* Parameterübergabe,
 - **nichtstriktheit** Funktionen
- **Haskell** ist
 - als **nicht-strikt** **spezifiziert**
 - (meist) mit **verzögerter Auswertung** **implementiert**.

Verzögerte Auswertung: Beispiele

- Beispiel:

```
second (fac 4) (fac 3)
```

Verzögerte Auswertung: Beispiele

- Beispiel:

`second (fac 4) (fac 3) \rightsquigarrow fac 3`

Verzögerte Auswertung: Beispiele

- Beispiel:

`second (fac 4) (fac 3) \rightsquigarrow fac 3 \rightsquigarrow 6`

`3+ second undefined 4`

Verzögerte Auswertung: Beispiele

- Beispiel:

`second (fac 4) (fac 3) \rightsquigarrow fac 3 \rightsquigarrow 6`

`3+ second undefined 4 \rightsquigarrow 3+ 4`

Verzögerte Auswertung: Beispiele

- Beispiel:

`second (fac 4) (fac 3) \rightsquigarrow fac 3 \rightsquigarrow 6`

`3+ second undefined 4 \rightsquigarrow 3+ 4 \rightsquigarrow 7`

- Erstes Argument von `second` wird nicht ausgewertet.
- Zweites Argument von `second` wird erst im Funktionsrumpf ausgewertet.

Datenorientierte Programme

Nichtstriktheit führt zur Datenorientierung

- Berechnung der Summe der Quadrate von 1 bis n:

```
sqrsum n = sum (map (^2) [1..n])
```

Datenorientierte Programme

Nichtstriktheit führt zur Datenorientierung

- Berechnung der Summe der Quadrate von 1 bis n:

```
sqrsum n = sum (map (^2) [1..n])
```

⇒ Es werden keine Listen von Zwischenergebnissen erzeugt.

Datenorientierte Programme

Nichtstriktheit führt zur **Datenorientierung**

- Berechnung der Summe der Quadrate von 1 bis n :

```
sqrsum n = sum (map (^2) [1..n])
```

⇒ Es werden **keine Listen** von **Zwischenergebnissen** erzeugt.

- Minimum einer Liste durch

```
min' :: Ord a => [a] -> a  
min' xs = head (msort xs)
```

Datenorientierte Programme

Nichtstriktheit führt zur Datenorientierung

- Berechnung der Summe der Quadrate von 1 bis n:

```
sqrsum n = sum (map (^2) [1..n])
```

⇒ Es werden **keine Listen** von **Zwischenergebnissen** erzeugt.

- Minimum einer Liste durch

```
min' :: Ord a => [a] -> a  
min' xs = head (msort xs)
```

⇒ Liste wird **nicht vollständig sortiert** — **binäre Suche**.

Unendliche Datenstrukturen: Ströme

- **Ströme** sind **unendliche Listen**.
 - **Unendliche** Liste `[2,2,2,...]`
`twos = 2 : twos`
 - Liste der **natürlichen Zahlen**:
`nat = [1..]`
 - Bildung von **unendlichen Listen**:
`cycle :: [a] -> [a]`
`cycle xs = xs ++ cycle xs`
- **Repräsentation** durch endliche, zyklische Datenstruktur
 - Kopf wird nur **einmal** ausgewertet.
- **Nützlich** für Listen mit **unbekannter Länge**

Zeigen.

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?
- Lösung: Berechnung **aller** Primzahlen, davon die **ersten** n .

```
sieve :: [Integer] -> [Integer]
sieve (p:ps) =
    p:(sieve (filter (\n-> n `mod` p /= 0) ps))
```

- **Keine** Rekursionsverankerung (vgl. **alte Version**)

```
primes :: [Integer]
primes = sieve [2..]
```

- Von allen Primzahlen die **ersten**:

```
nprimes :: Int -> [Integer]
nprimes n = take n primes
```

Testen.

Bsp: Fibonacci-Zahlen

- Aus der Kaninchenzucht.
- Sollte jeder Informatiker kennen.

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- **Problem:** exponentieller Aufwand.

Bsp: Fibonacci-Zahlen

- **Lösung:** zuvor berechnete **Teilergebnisse wiederverwenden.**
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

<code>fibs</code>	1	1	2	3	5	8	13	21	34	55
<code>tail fibs</code>	1	2	3	5	8	13	21	34	55	
<code>tail (tail fibs)</code>	2	3	5	8	13	21	34	55		

- Damit ergibt sich:

Bsp: Fibonacci-Zahlen

- **Lösung:** zuvor berechnete **Teilergebnisse wiederverwenden**.
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
      fibs  1  1  2  3  5  8 13 21 34 55
    tail fibs 1  2  3  5  8 13 21 34 55
tail (tail fibs) 2  3  5  8 13 21 34 55
```

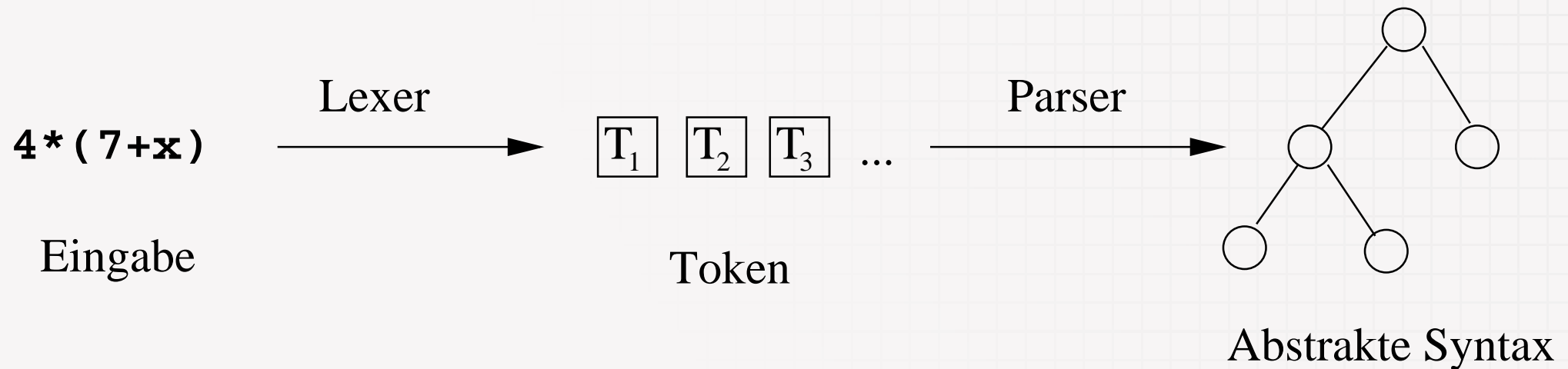
- Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- `n`-te Fibonaccizahl mit `fibs !! n` Zeigen.
- **Aufwand: linear**, da `fibs` nur einmal ausgewertet wird.

Fallstudie: Parsierung

- Gegeben: **Grammatik**
- Gesucht: Funktion, die **Wörter** der Grammatik **erkennt**



Parser

- **Parser** bilden Eingabe auf Parsierungen ab
 - Mehrere Parsierungen möglich
 - Backtracking möglich
 - Durch verzögerte Auswertung dennoch effizient
- **Basisparser** erkennen **Terminalsymbole**
- **Parserkombinatoren** erkennen **Nichtterminalsymbole**
 - Sequenzierung (erst A , dann B)
 - Alternierung (entweder A oder B)
 - Abgeleitete Kombinatoren (z.B. **Listen** A^* , **nicht-leere Listen** A^+)

Grammatik für Arithmetische Ausdrücke

$Expr ::= Term + Term \mid Term - Term \mid Term$

$Term ::= Factor * Factor \mid Factor / Factor \mid Factor$

$Factor ::= Number \mid Variable \mid (Expr)$

$Number ::= Digit^+$

$Digit ::= 0 \mid \dots \mid 9$

$Var ::= Char^+$

$Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik **abstrakte Syntax** (siehe VL 05.12.06)

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik **abstrakte Syntax** (siehe VL 05.12.06)

```
data Expr    = Plus    Expr Expr
              | Minus   Expr Expr
              | Times   Expr Expr
              | Div     Expr Expr
              | Number  Int
              | Var     String
              deriving (Eq, Show)
```

- Hier Unterscheidung Term, Factor, Number unnötig.

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über **Eingabetyp** (Token) **a** und **Ergebnis** **b**

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über **Eingabetyp** (Token) a und **Ergebnis** b
- Müssen **mehrdeutige Ergebnisse** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über **Eingabetyp** (Token) **a** und **Ergebnis** **b**
- Müssen **mehrdeutige Ergebnisse** modellieren
- Müssen **Rest der Eingabe** modellieren

Modellierung in Haskell

Welcher **Typ** für Parser?

- Parser übersetzt **Token** in **abstrakte Syntax**
- Parametrisiert über **Eingabetyp** (Token) **a** und **Ergebnis** **b**
- Müssen **mehrdeutige Ergebnisse** modellieren
- Müssen **Rest der Eingabe** modellieren

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

Basisparser

- Erkennt **nichts**:

```
none :: Parse a b
none = const []
```

- Erkennt **alles**:

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

- Erkennt **einzelne Zeichen**:

```
token :: Eq a => a -> Parse a a
token t = spot (== t)
spot :: (a -> Bool) -> Parse a a
spot p [] = []
spot p (x:xs) = if p x then [(x, xs)] else []
```

Basiskombinatoren

- Alternierung:

```
infixl 3 'alt'
```

```
alt :: Parse a b-> Parse a b-> Parse a b
```

```
alt p1 p2 i = p1 i ++ p2 i
```

- Sequenzierung:

- Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
```

```
(>*>) :: Parse a b-> Parse a c-> Parse a (b, c)
```

```
(>*>) p1 p2 i = [((y, z), r2) | (y, r1) <- p1 i,  
                                (z, r2) <- p2 r1]
```

Basiskombinatoren

- **Ausgabe** weiterverarbeiten:

```
infix 4 'use'
use :: Parse a b-> (b-> c)-> Parse a c
use p f inp = [(f x, r) | (x, r)<- p inp]
```

- Damit z.B. Sequenzierung **rechts/links**:

```
infixl 5 *>, >*
(*>) :: Parse a b-> Parse a c-> Parse a c
(>*) :: Parse a b-> Parse a c-> Parse a b
p1 *> p2 = p1 >*> p2 'use' snd
p1 >* p2 = p1 >*> p2 'use' fst
```

Abgeleitete Kombinatoren

- **Listen:** $A^* ::= AA^* \mid \varepsilon$

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

- **Nicht-leere Listen:** $A^+ ::= AA^*$

```
some :: Parse a b-> Parse a [b]
some p = p >*> list p 'use' uncurry (:)
```

- NB. Präzedenzen: $>*>$ (5) vor `use` (4) vor `alt` (3)

Einschub: Präzedenzen der Operatoren in Haskell

Höchste Priorität: Funktionsapplikation

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, 'quot', 'rem', 'div', 'mod'
infixl 6  +, -
infixr 5  :
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, 'seq'
```


Parsierung Arithmetischer Ausdrücke

- Token: Char
- Parsierung von Expr

```
pExpr :: Parse Char Expr
```

```
pExpr = pTerm >* token '+' >*> pTerm 'use' uncurry Plus  
      'alt' pTerm >* token '-' >*> pTerm 'use' uncurry Minus  
      'alt' pTerm
```

- Parsierung von Term

```
pTerm :: Parse Char Expr
```

```
pTerm = pFactor >* token '*' >*> pFactor 'use' uncurry Times  
      'alt' pFactor >* token '/' >*> pFactor 'use' uncurry Div  
      'alt' pFactor
```

Parsierung Arithmetischer Ausdrücke

- Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isDigit) 'use' Number. read
        'alt' some (spot isAlpha) 'use' Var
        'alt' token '(' *> pExpr >* token ')'
```

Die Hauptfunktion

- Lexing: **Leerzeichen** aus der Eingabe **entfernen**
- Zu prüfen:
 - Parsierung **konsumiert** Eingabe
 - Keine **Mehrdeutigkeit**

Testen.

```
parse :: String -> Expr
parse i =
  case filter (null . snd)
    (pExpr (filter (not.isSpace) i)) of
    [] -> error "Input does not parse."
    [(e, _)] -> e
    _ -> error "Input is ambiguous."
```

Ein kleiner Fehler

- **Mangel:** 3+4+5 führt zu **Syntaxfehler** — Fehler in der **Grammatik**
- Behebung: **Änderung** der Grammatik

$$\begin{aligned} \text{Expr} &::= \text{Term} + \text{Expr} \mid \text{Term} - \text{Expr} \mid \text{Term} \\ \text{Term} &::= \text{Factor} * \text{Term} \mid \text{Factor} / \text{Term} \mid \text{Factor} \\ \text{Factor} &::= \text{Number} \mid \text{Variable} \mid (\text{Expr}) \\ \text{Number} &::= \text{Digit}^+ \\ \text{Digit} &::= 0 \mid \dots \mid 9 \\ \text{Var} &::= \text{Char}^+ \\ \text{Char} &::= a \mid \dots \mid z \mid A \mid \dots \mid Z \end{aligned}$$

- (vergleiche alt)
- **Abstrakte Syntax** bleibt

Änderung des Parsers

- Entsprechende Änderung des Parsers in `pExpr`

```
pExpr :: Parse Char Expr
```

```
pExpr = pTerm >* token '+' >*> pExpr 'use' uncurry Plus  
      'alt' pTerm >* token '-' >*> pExpr 'use' uncurry Minus  
      'alt' pTerm
```

- ... und in `pTerm`:

```
pTerm :: Parse Char Expr
```

```
pTerm = pFactor >* token '*' >*> pTerm 'use' uncurry Times  
      'alt' pFactor >* token '/' >*> pTerm 'use' uncurry Div  
      'alt' pFactor
```

- (vergleiche alt)

- `pFactor` und Hauptfunktion bleiben.

Testen.

Zusammenfassung Parserkombinatoren

- **Systematische Konstruktion** des Parsers aus der Grammatik.
- **Abstraktion** durch Funktionen höherer Ordnung.
- Durch verzögerte Auswertung **annehmbare Effizienz**:
 - Grammatik muß **eindeutig** sein (LL(1) o.ä.)
 - Vorsicht bei **Mehrdeutigkeiten**!
 - Effizient implementierte **Büchereien** mit **gleicher Schnittstelle** auch für **große Eingaben** geeignet.

Zusammenfassung

- **Verzögerte Auswertung** erlaubt **unendliche Datenstrukturen**
 - Zum Beispiel: Ströme (unendliche Listen)
- **Parserkombinatoren:**
 - **Systematische Konstruktion** von Parseern
 - Durch verzögerte Auswertung **annehmbare Effizienz**

Zusammenfassung

- **Verzögerte Auswertung** erlaubt **unendliche Datenstrukturen**
 - Zum Beispiel: Ströme (unendliche Listen)
- **Parserkombinatoren:**
 - **Systematische Konstruktion** von Parsern
 - Durch verzögerte Auswertung **annehmbare Effizienz**

Fröhliche Weihnachten!

