

Praktische Informatik 3
Einführung in die Funktionale Programmierung
Vorlesung vom 03.12.08:
Abstrakte Datentypen

Christoph Lüth

WS 08/09



Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
 - Abstrakte Datentypen
 - Signaturen & Axiome
 - Korrektheit von Programmen
 - Zustand und Aktionen
- Teil III: Beispiele, Anwendungen, Ausblicke

Inhalt

- Abstrakte Datentypen
 - Was ist das?
 - Wozu braucht man das?
- Fallbeispiel: Parserkombinatoren

Einfache Bäume

- Letzte Vorlesung: Bäume

```
data Tree a = Null
            | Node (Tree a) a (Tree a)
```
- Dazu Test auf Enthaltensein:

```
member :: Eq a => Tree a -> a -> Bool
member Null _ = False
member (Node l a r) b =
  a == b || (member l b) || (member r b)
```
- Problem: Suche aufwändig (Backtracking)
- Besser: Baum geordnet
 - Noch besser: Baum balanciert etc.

Geordnete Bäume

- Voraussetzung:
 - Ordnung auf a (Ord a)
 - Es soll für alle Bäume gelten:
$$\forall x \ t. \ t = \text{Node } l \ a \ r \implies (\text{member } x \ l \implies x < a) \wedge (\text{member } x \ r \implies a < x)$$
- Test auf Enthaltensein vereinfacht:

```
member :: Ord a => Tree a -> a -> Bool
member Null _ = False
member (Node l a r) b
  | b < a = member l b
  | a == b = True
  | b > a = member r b
```

Geordnete Bäume

- Ordnungserhaltendes Einfügen

```
insert :: Ord a => Tree a -> a -> Tree a
insert Null a = Node Null a Null
insert (Node l a r) b
  | b < a = Node (insert l b) a r
  | b == a = Node l a r
  | b > a = Node l a (insert r b)
```
- Problem: Erzeugung ungeordneter Bäume möglich.
- Lösung: Verstecken der Konstrukturen.

Geordnete Bäume als abstrakter Datentyp

- Es gibt einen Typ `Tree a`
- Es gibt Operationen
 - `insert :: Ord a => Tree a -> a -> Tree a`
 - `member :: Ord a => Tree a -> a -> Bool`
 - `empty :: Ord a => Tree a`
 - ... und keine weiteren!
- Beispiel für einen abstrakten Datentypen
 - Typ und Operationen darauf
 - Operationen charakterisieren den Typen

ADTs in Haskell: Module

- Einschränkung der Sichtbarkeit durch Verkapselung
- Modul: Kleinste verkapselbare Einheit
- Ein Modul umfaßt:
 - Definitionen von Typen, Funktionen, Klassen
 - Deklaration der nach außen sichtbaren Definitionen
- Syntax:

```
module Name (sichtbare Bezeichner) where Rumpf
```

 - sichtbare Bezeichner können leer sein
 - Gleichzeitig: Übersetzungseinheit (getrennte Übersetzung)

Geordnete Bäume als ADT

```

module OrdTree(
  Tree
  , empty
  , insert
  , member ) where

data Tree a = Null
  | Node (Tree a) a (Tree a)
  deriving Show

empty :: Ord a => Tree a
empty = Null

... sowie member und insert

```

Benutzung von ADTs

- **Operationen** und **Typen** müssen bekannt gemacht werden (Import)
- Möglichkeiten des Imports:
 - **Alles** importieren
 - Nur bestimmte Operationen und Typen importieren
 - Bestimmte Typen und Operationen nicht importieren

Importe in Haskell

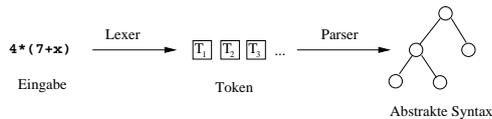
- Schlüsselwort: `import Name [hiding] (Bezeichner)`
- *Bezeichner* geben an, **was** importiert werden soll:
 - Ohne Bezeichner wird **alles** importiert
 - Mit `hiding` werden Bezeichner **nicht** importiert
 - Alle Importe stehen immer am **Anfang** des Moduls

Beispiel: Importe von Bäumen

Import(e)	Bekannte Bezeichner
<code>import OrdTree</code>	<code>Tree, insert, member, empty</code>
<code>import OrdTree(Tree, empty)</code>	<code>Tree, empty</code>
<code>import OrdTree(insert)</code>	<code>insert</code>
<code>import OrdTree hiding (member)</code>	<code>Tree, empty, insert</code>
<code>import OrdTree(empty)</code>	<code>empty,</code>
<code>import OrdTree hiding (empty)</code>	<code>Tree, insert, member</code>

Einführung: Parser

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



Parser

- **Parser** bilden Eingabe auf Parsierungen ab
 - Mehrere Parsierungen möglich
 - Backtracking möglich
- **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 | Term
Term ::= Factor * Factor | Factor
Factor ::= Variable | (Expr)
Variable ::= Char+
Char ::= a | ... | z | A | ... | Z

```

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik **abstrakte Syntax**
- ```

data Expr = Plus Expr Expr
 | Times Expr Expr
 | Var String
 deriving (Eq, Show)

```
- Hier Unterscheidung **Term, Factor, Number** unnötig.

## Modellierung in Haskell

Welcher **Typ** für Parser?

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

- Parametrisiert über **Eingabetyp** (Token) *a* und **Ergebnis** *b*
- Parser übersetzt Token in abstrakte Syntax
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren
- Beispiel: "a+b\*c"  $\rightsquigarrow$  [  
 (Var "a", "+b\*c"),  
 (Plus (Var "a") (Var "b") , "\*c"),  
 (Plus (Var "a") (Times (Var "b") (Var "c")), "")]



## Basisparser

- Erkennt **nichts**:  
 none :: Parse a b  
 none = const []
- Erkennt **alles**:  
 succeed :: b-> Parse a b  
 succeed b inp = [(b, inp)]
- Erkennt **einzelne Token**:  
 spot :: (a-> Bool)-> Parse a a  
 spot p [] = []  
 spot p (x:xs) = if p x then [(x, xs)] else []
- token :: Eq a => a-> Parse a a  
 token t = spot (\c-> t == c)
- Warum nicht none, succeed durch spot? Typ!



## Basiskombinatoren: alt, >\*>

- **Alternierung**:
    - Erste Alternative wird bevorzugt
- ```
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 =
 concatMap (\(b, r)->
 map (\(c, s)-> ((b, c), s)) (p2 r)) (p1 i)
```



## Basiskombinatoren: use

- **Rückgabe** weiterverarbeiten:
- ```
infix 4 'use', 'use2'  
use :: Parse a b-> (b-> c)-> Parse a c  
use p f i = map (\(o, r)-> (f o, r)) (p i)
```
- ```
use2 :: Parse a (b, c)-> (b-> c-> d)-> Parse a d
use2 p f = use p (uncurry f)
```
- 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**: $A^* ::= AA^* | \epsilon$
- ```
list :: Parse a b-> Parse a [b]
list p = p >*> list p 'use2' (:)
 'alt' succeed []
```
- **Nicht-leere Listen**:  $A^+ ::= AA^*$
- ```
some :: Parse a b-> Parse a [b]  
some p = p >*> list p 'use2' (:)
```
- NB. Präzedenzen: >*> (5) vor use (4) vor alt (3)



Schnittstelle

- Nach außen nur Typ **Parse** sichtbar, plus Operationen darauf:
- ```
module ParserCombinators(
 Parse
 , none , succeed , token , spot
 , (>*>) , use , use2 , (*>) , (>*) , alt
 , list , some
) where ...
```
- Struktur von **Parse** zur Benutzung irrelevant
    - Grammatik sollte **eindeutig** sein (LL/LR(1) o.ä.)
    - Vorsicht bei **Mehrdeutigkeiten!**
    - Effizient implementierte Büchereien mit gleicher Schnittstelle auch für große Eingaben geeignet.



## Parsierung Arithmetischer Ausdrücke

- Token: Char
- Import:  
 import ParserCombinators
- Parsierung von Factor  
 pFactor :: Parse Char Expr  
 pFactor = some (spot isAlpha) 'use' Var  
 'alt' token '(' \*> pExpr >\*> token ')'
- Parsierung von Term  
 pTerm :: Parse Char Expr  
 pTerm = pFactor >\*> token '\*' >\*> pFactor 'use2' Times  
 'alt' pFactor
- Parsierung von Expr  
 pExpr :: Parse Char Expr  
 pExpr = pTerm >\*> token '+' >\*> pTerm 'use2' Plus  
 'alt' pTerm



## Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
  - Zu prüfen:
    - Parsierung konsumiert Eingabe
    - Keine Mehrdeutigkeit
- ```
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
 - $Expr ::= Term + Expr \mid Term$
 - $Term ::= Factor * Term \mid Factor$
 - $Factor ::= Variable \mid (Expr)$
 - $Variable ::= Char^+$
 - $Char ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$
- **Abstrakte Syntax** bleibt



Änderung des Parsers

- Entsprechende **Änderung** des Parsers in `pExpr`

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr 'use2' Plus
      'alt' pTerm
```

- ... und in `pTerm`:

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pTerm 'use2' Times
      'alt' pFactor
```

- `pFactor` und **Hauptfunktion** bleiben.



Zusammenfassung

- **Abstrakte Datentypen (ADTs):**
 - Besteht aus **Typ** und **Operationen** darauf
- Realisierung in Haskell durch **Module**
- Fallbeispiel Parserkombinatoren:
- **Systematische Konstruktion** des Parsers aus der Grammatik
- **Abstraktion** durch Funktionen höherer Ordnung
- **Verkapselung** als ADT

