Praktische Informatik 3: Funktionale Programmierung

Vorlesung 12 / 23. Januar 2024

Parsing & Domain-Specific Languages

Thomas Barkowsky

Wintersemester 2023/24

Universität
Bremen

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- I/O, Aktionen und Zustände
- Testen und Qualitätssicherung
- Monaden
- Parsing & domänenspezifische Sprachen

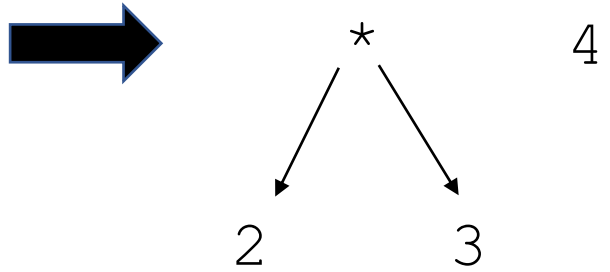
heute in dieser Vorlesung...

- Parser
  - Grundlagen
  - Parser bauen in Haskell
  - Beispiel: Parsen arithmetischer Ausdrücke
- Domänenspezifische Sprachen
  - Definition und Eigenschaften
  - Beispiel: reguläre Ausdrücke

Parser

Was ist ein Parser?

- Ein Parser wandelt eine Zeichenkette um in eine eindeutige syntaktische Struktur
  - in der Regel ein Baum, z.B.: `2 * 3 + 4`



  - Operatoren an Knoten, Operanden in Blättern
  - explizit: Stelligkeit der Operanden, Priorisierung
- Parser essentiell für die Verarbeitung (komplexer) Eingaben
  - Strukturierung von Eingaben
  - Vorbereitung für weitere Verarbeitung

Parser als Funktionen

- z.B. so:
- `type Parser = String -> Tree`
  - für einen passend strukturierten `Tree`
- allerdings...
  - ein einzelner Parser wird evtl. nur einen Teil des Strings übersetzen
  - ein Parser könnte auch scheitern oder mehr als ein Ergebnis liefern
  - der Rückgabewert eines Parsers muss nicht zwingend ein Baum sein
- somit besser...

Verallgemeinerte Parser-Funktion

- `type Parser a = String -> [(a, String)]`
  - erhält Zeichenkette als Eingabe
  - liefert Rückgabewert vom Typ a
  - als Paar zusammen mit einem `String`
  - in Form einer Liste (kann bei fehlgeschlagener Anwendung leer sein)
- und das erinnert uns an was?
  - ... den *state transformer* von letzter Woche: `State -> (a, State)`
  - der zu verändernde Zustand ist hierbei vom Typ `String`
  - Rückgabe als Liste: es kann mehr als ein Zustand (oder gar keiner) erzeugt werden
- damit: Parser als verallgemeinerter *state transformer*...

Monaden für Zustände

(aus Vorlesung 11)

- Funktionen, die mit veränderlichen Zuständen umgehen
- Vereinfachende Annahme (o.B.d.A.): Zustände als `Int` Werte:

```
type State = Int
```

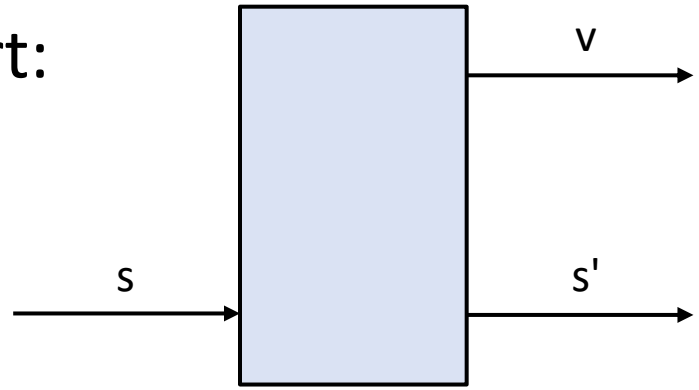
- Funktion zur Veränderung eines Zustands (*state transformer*):

```
type ST = State -> State
```

- besser: *state transformer* mit Rückgabewert:

```
type ST a = State -> (a, State)
```

  - z.B. durchlaufende Buchungsnummer und Kontostand
  - Wie kommt der Rückgabewert zustande? ...



Definition des Parsers (Basis)

- Vorgehen analog zu *state transformer*
- Re-Definition mit `newtype` (für Klassen-Instantiierung):

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

  - `P` als Dummy-Konstruktor
- Anwendung des Parsers auf eine Eingabezeichenkette:

```
parse :: Parser a -> String -> [(a, String)]
parse (P p) inp = p inp
```

- unser erster Parser...

Ein erster Parser…

- Soll als Grundlage für alle weiteren Parser dienen
- Extrahiert erstes Zeichen aus Eingabestring, scheitert bei leerem Eingabestring:

```
item :: Parser Char
item = P (\inp -> case inp of
                    []     -> []
                    (x:xs) -> [(x, xs)])
```

```
> parse item ""
[]
> parse item "abc"
[('a',"bc")]
```

`Parser` als `Functor`

- `fmap`: Funktionsanwendung auf Parsing-Ergebnis bei Erfolg, sonst Propagation der leeren Liste:

```
instance Functor Parser where
   -- fmap :: (a -> b) -> Parser a -> Parser b
   fmap g p = P (\inp -> case parse p inp of
                           []         -> []
                           [(v,out)] -> [(g v, out)])
```

```
> parse (fmap toUpper item) "abc"
[('A',"bc")]
> parse (fmap toUpper item) ""
[]
```

(`toUpper` importiert aus `Data.Char`)

`Parser` als `Applicative` Functor

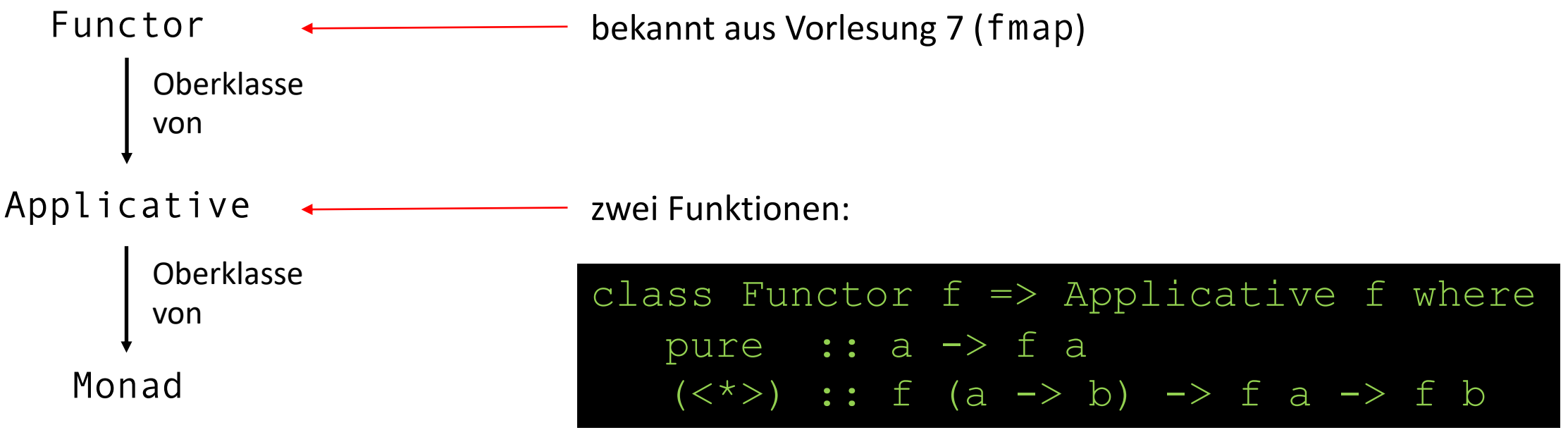
```
instance Applicative Parser where
   -- pure :: a -> Parser a
   pure v = P (\inp -> [(v,inp)])
   -- <*> :: Parser (a -> b) -> Parser a -> Parser b
   pg <*> px = P (\inp -> case parse pg inp of
                            []        -> []
                            [(g,out)] -> parse (fmap g px) out)
```

- `pure` wandelt Wert in `Parser` um
  - immer erfolgreich, ohne Veränderung des Eingabestrings
- `<*>` wendet `Parser` mit Rückgabe einer Funktion (1) auf `Parser` mit Rückgabe eines Wertes (2) an, dann Anwendung (2) auf (1)
  - erfolgreich, wenn alle Komponenten erfolgreich

Was noch fehlt…

(aus Vorlesung 11)

- Typklassen-Hierarchie seit GHC 7.1 (2015):
  - nicht in *Haskell 2010 Language Report*




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

`Parser` als `Applicative` Functor (2)

- Beispiel: Parser, der drei Zeichen einliest, das zweite ignoriert und das erste und dritte als Paar zurückgibt:

```
three :: Parser (Char, Char)
three = pure g <*> item <*> item <*> item
        where g x y z = (x, z)
```

```
> parse three "abcdef"
[(('a','c'),"def")]
```

  - Misserfolg, wenn Eingabestring zu kurz:

```
> parse three "ab"
[]
```

`Parser` als Monade

```
instance Monad Parser where
   -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
   p >>= f = P (\inp -> case parse p inp of
                          []        -> []
                          [(v,out)] -> parse (f v) out)
```

- *bind*-Operator erzeugt `Parser` durch Anwendung von `p` auf Eingabestring, dann Anwendung von Funktion `f` auf dessen Ergebnis
  - Misserfolg wenn Anwendung von `p` fehlschlägt
- damit Formulierung von `three` in `do`-Notation:

```
three :: Parser (Char, Char)
three = do x <- item
           item
           z <- item
           return (x, z)
```

Arbeiten mit Parsern

Kombination von Parsern

- Mit `do`-Notation können wir Parser sequenzieren
  - Rückgabe eines Parsers wird an nachfolgenden übergeben
- Auch nützlich: Alternativen!
  - Wenn Parser 1 scheitert, dann wird Parser 2 auf dieselbe Eingabe angewendet
  - Verwendung der Klasse `Alternative` (`import Control.Applicative`):

```
class Applicative f => Alternative f where
   empty :: f a
   (<|>) :: f a -> f a -> f a
```

  - zwei Funktionen:
    - `empty`: Misserfolg
    - `<|>`: Auswahl-Operator

Die Klasse `Alternative`

- Identität und Assoziativität:

```
empty <|> x       = x
x <|> empty       = x
x <|> (y <|> z) = (x <|> y) <|> z
```

- Beispiel `Maybe`:

```
instance Alternative Maybe where
   empty             = Nothing
   Nothing  <|> my = my
   (Just x) <|> _    = Just x
```

  - `Nothing` als Misserfolg
  - bei Auswahl: erstes Argument, wenn nicht `Nothing`, sonst zweites Argument

- sehr ähnlich für Parser: ...

`Alternative` für Parser

```
instance Alternative Parser where
   empty   = P (\inp -> [])
   p <|> q = P (\inp -> case parse p inp of
                          []         -> parse q inp
                          [(v,out)] -> [(v,out)])
```

- `empty` erzeugt einen Parser, der immer fehlschlägt
- `<|>` gibt bei Erfolg Ergebnis des ersten Parsers zurück, sonst das des zweiten
- Beispiele: ...

`Alternative` für Parser: Beispiele

```
> parse empty "abc"
[]
> parse (three <|> return ('x', 'y')) "abc"
[(('a','c'),"")]
> parse (three <|> return ('x', 'y')) "ab"
[(('x','y'),"ab")]
```

- Was brauchen wir noch?
  - ... z.B. einen Parser der ein Prädikat auf ein Zeichen anwendet:

Parser: Prädikat für Zeichen

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else empty
```

- Beispiel:

```
digit :: Parser Char
digit = sat isDigit
```

```
> parse digit "123"
[('1',"23")]
> parse digit "abc"
[]
```

Parser für Zeichen und Strings

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

```
> parse (char 'a') "abc"
[('a',"bc")]
```

```
string :: String -> Parser String
string []     = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

```
*Main> parse (string "abc") "abcdef"
[("abc","def")]
```

`many p` und `some p`

- ebenfalls definiert in Klasse `Alternative`
- Anwendung eines Parsers solange bis er fehlschlägt
  - many: keine oder mehrfache Anwendung
  - some: mindestens einmalige Anwendung

```
> parse (many digit) "123abc"
[("123","abc")]
> parse (many digit) "abc"
[("","abc")]
> parse (some digit) "abc"
[]
```

Parser mit `many` und `some`

- Parser für Wörter, die mit Kleinbuchstaben beginnen:

```
ident :: Parser String
ident = do x  <- lower
           xs <- many alphanum
           return (x:xs)
```

- mit:

```
lower :: Parser Char
lower = sat isLower

alphanum :: Parser Char
alphanum = sat isAlphaNum
```

- Beispiel:

```
> parse ident "caMel nextWord"
[("caMel"," nextWord")]
```

Parser für Zahlen

```
nat :: Parser Int
nat = do xs <- some digit
         return (read xs)

int :: Parser Int
int = do char '-'
         n <- nat
         return (-n)
      <|> nat
```

```
> parse nat "123 abc"
[(123," abc")]


> parse int "-123 abc"
[(-123," abc")]
```

- `nat` konvertiert Ergebnis nach `Int`

Parser für Leerzeichen

```
space :: Parser ()
space = do many (sat isSpace)
           return ()
```

```
> parse space "   Wort"
[((),"Wort")]
```

- Parser konsumiert Leerzeichen
- Rückgabe des leeren Tupels als Dummy
- Aber Leerzeichen sind ja typischerweise uninteressant...

Leerzeichen ignorieren

- ... vor und nach der Anwendung eines Parsers für ein *Token*:

```
token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
```

- damit:

```
natural :: Parser Int
natural = token nat

symbol :: String -> Parser String
symbol xs = token (string xs)
```

- Beispiel zu `symbol`: ...

Leerzeichen ignorieren in Listen von Zahlen

```
nats :: Parser [Int]
nats = do symbol "["
          n  <- natural
          ns <- many (do symbol ","
                         natural)
          symbol "]"
          return (n:ns)
```

- Anwendungsbeispiel: arithmetische Ausdrücke parsen ...

Anwendung: Parsen arithmetischer Ausdrücke

Parsen (einfacher) arithmetischer Ausdrücke

- Natürliche Zahlen, Addition, Multiplikation, Klammern
- Multiplikation höhere Priorität als Addition, sonst rechtsassoziativ
- Grammatik:

```
expr   ::= term [ + expr | ε ]
term   ::= factor [ * term | ε ]
factor ::= ( expr ) | nat
nat    ::= 0 | 1 | 2 | 3 | …
```

- Übersetzen der Grammatik in Parser unter Verwendung der zuvor definierten Parser ...

Der Arithmetikparser:

- Alle Parserfunktionen liefern Ergebnisse der Rechnungen zurück (keinen Syntaxbaum)

- ... mit einer netten kleinen Auswertungsfunktion: ...

```
expr :: Parser Int
expr = do t <- term
          do symbol "+"
             e <- expr
             return (t + e)
           <|> return t

term :: Parser Int
term = do f <- factor
          do symbol "*"
             t <- term
             return (f * t)
           <|> return f

factor :: Parser Int
factor = do symbol "("
            e <- expr
            symbol ")"
            return e
          <|> natural
```

`eval` für Arithmetikparser

```
eval :: String -> Int
eval xs = case (parse expr xs) of
            [(n,[])]  -> n
            [(_,out)] -> error ("Unused input " ++ out)
            []        -> error "Invalid input"
```

```
> eval "2*3+4"
10
> eval "2*(3+4)"
14
> eval "2*3^4"
*** Exception: Unused input ^4
> eval "one plus two"
*** Exception: Invalid input
```

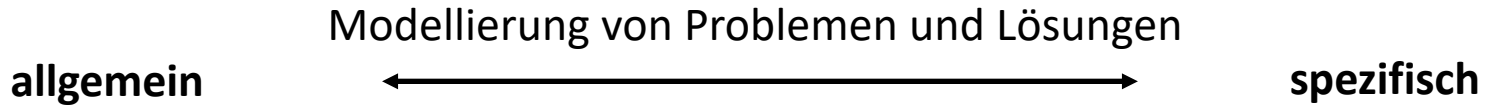
Domänenspezifische Sprachen

Programmiersprachen

- *General-Purpose Languages* (GPL) vs. *Domain-Specific Languages* (DSL)
- DSL: Beispiele
  - Textverarbeitungssprachen (LaTeX, HTML): Layout-Beschreibung für Text
  - Hardwaredesign-Sprachen (VHDL, Lava): Transisoren, Gatter, Signalströme, etc.
  - Grafiksprachen (SVG, PDF): Beschreibung grafischer Elemente
  - Tabellenkalkulation (Excel, Numbers): Numerische Rechnungen (funktional!)
  - Datenbanksprachen (SQL): Datenstrukturen, z.B. relational
  - Steuerungssprachen, z.B. Robotik, techn. Anlagen
- Viele ähnliche Features bei GPL und DSL
  - z.B. Datentypen, Strukturiertheit, Gliederung, Sub-Komponenten, ...

GPLs vs. DSLs

- GPL
  - (Turing-)Mächtigkeit
  - Für viele Problemklassen nutzbar
  - Abstrakt bzgl. eines Problems
- DSL
  - Mächtigkeit oft reduziert
  - maßgeschneidert für spezifische Probleme
  - geringer Abstand zum Problem
  - oft Teil einer Programmiersprache, eingebettet oder alleinstehend



Definition und Eigenschaften

- Definition: "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."
  (van Deursen et al., 2000)
- Eigenschaften
  - Fokussierte Ausdrucksmächtigkeit (evtl. nicht Turing-mächtig)
  - oft klein (*micro-languages*), eingeschränkte Anzahl von Sprachkonstrukten
  - Domäneneigenschaften in Sprachkonstrukten und Vokabular
  - oft deklarativ

Stand-alone vs. eingebettete DSLs

- *Stand-alone* DSL:
  - Programmiersprache beliebig
  - vollständige Kontrolle über Syntax und Semantik
  - hoher Aufwand, da komplette Neuentwicklung
- Eingebettete DSL:
  - setzt auf vorhandener Programmiersprache (hier Haskell) auf
  - schnelle Realisierung, aber Abhängigkeit von vorhandener Semantik
  - funktionale Programmiersprachen hierfür sehr gut geeignet
    - algebraische Datentypen, Funktionen höherer Ordnung (Kombinatoren!), etc.
  - spätere Erweiterung zu *stand-alone* DSL möglich

Flache vs. tiefe Einbettung

- Flache Einbettung
  - direkte Realisierung von Domänenfunktionen
  - Objekte der Domäne nicht explizit in Haskell repräsentiert
  - schnelle Realisierbarkeit
- Tiefe Einbettung
  - Domänenobjekte als Haskell-Datentyp (i.d.R. ADT)
  - Domänenfunktionen auf definiertem Typen
  - Reichhaltigere Funktionalität auf ADT (Manipulation, Analyse, etc.)

- Beispiel für flach eingebettete DSL ...

Beispiel: Prüfen regulärer Ausdrücke

- Reguläre Ausdrücke beschreiben Mengen von Zeichenketten
- Reguläre Ausdrücke sind
  - der leere String ε
  - ein einzelnes Zeichen x (aus einem zugrundeliegenden Alphabet)
  - $(r_1|r_2)$: alternativ r_1 oder r_2; r_1, r_2 reguläre Ausdrücke
  - (r_1r_2): Sequenzierung von r_1 und r_2
  - $(r_1)^*$: Iterierung von r_1

Beispiel: Prüfen regulärer Ausdrücke (2)

```
type RegExpr = String -> Bool

epsilon :: RegExpr

char    :: Char -> RegExpr

(||||)  :: RegExpr -> RegExpr -> RegExpr

(<**>)  :: RegExpr -> RegExpr -> RegExpr

star    :: RegExpr -> RegExpr
```

Datentyp als Funktion

elementare reguläre Ausdrücke

Kombinatoren

- Demo!

Tiefe Einbettung kann mehr!

- Eigener Datentyp (ADT) für reguläre Ausdrücke
- Eigene Syntax und Semantik (z.B. Priorität der Operatoren)
- Benennung von Objekten
- Zustände und Seiteneffekte
- etc.

DSLs: Vor- und Nachteile

+

- problemspezifische Lösung
- domänenspezifische Notation
- knapp, oft selbstdokumentierend
- eingeschränkte Sprachmächtigkeit
- oft auch für Laien gut nutzbar

–

- Entwicklungskosten
- zusätzlicher Schulungsbedarf
- Sprachdesign i.A. nicht trivial
- i.d.R. geringes Toolangebot (Debugging, Dokumentation, etc.)
- evtl. Effizienzverlust

nächstes Mal...

- Rückblick
  - Wrap-up des Kurses
  - Bedeutung funktionaler Programmierung
- Ausblick E-Klausur
  - Kriterien und Anmeldung
  - Beispiele zur Vorbereitung