Mehr über Monaden

Heute gibt's:

- Monaden, mehr Monaden, und noch mehr Monaden,
- Kombination von Monaden,
- Fallbeispiel: monadische Parserkombinatoren.

Der Identitätsmonade

Der allereinfachste Monade:

```
type Id a = a

instance Monad Id where
  return a = a
  b >>= f  = f b
```

Der Fehlermonade

`Maybe` kennt nur `Nothing`, daher:

```
data Either a b = Left a  | Right b

instance Monad (Either String) where
   (Right a) >>= f     = f a
   (Left l)  >>= f     = Left l
   return b            = Right b
```

Der Listenmonade

Listen sind eine Instanz von `Monad`

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
    fail s   = []
```

Intuition: `f :: a-> [b]` Liste der möglichen Resultate

Der Lesemonade

```
newtype Reader e a = Reader (e -> a)

instance Monad (Reader e) where
    return a           = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> let Reader g = f (r e)
                                       in  g e
```

- Intuition: Werte der Eingabe e lesen und verabeiten.
- Eingabe wird nicht modifiziert.
    - Nicht der Fall bei Lesen aus Datei
- Beispiel: Lesen aus einer Symboltabelle

Der Schreibemonade

- Produziert einen Strom von Werten.

```
newtype Writer w a = Writer (a,[w])

instance Monad (Writer w) where
  return a             = Writer (a, [])
  (Writer (a,w)) >>= f = let Writer (a',w') = f a
                         in Writer (a', w++ w')
```

- Kein Zugriff auf geschriebene Werte möglich
- Beispiel: “Logging”

Der Continuationmonade

- Auswertungskontext wird explizit modelliert.

```
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
```

- `r` ist der Typ der gesamten Berechnung
- `a-> r` ist der momentane Kontext

```
instance Monad (Cont r) where
    return a       = Cont $ \k -> k a
    (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)
```

- `GOTO` für funktionale Sprachen.
- Lieber nicht benutzen.

Kombination von Monaden: Das Problem

Gegeben zwei Monaden:

```
class Monad m1 where ...
```

```
class Monad m2 where ...
```

Dann gelten weder

```
class Monad (m1 (m2 a))
class Monad (m2 (m1 a))
```

Problem: Monadengesetze gelten nicht.

Eine Lösung: Monadentransformer

- Monadentransformer: Monade mit "Loch"
- Beispiel: Zustandsmonadentransformer

```
type StateT s m a =  s-> m (s, a)
```

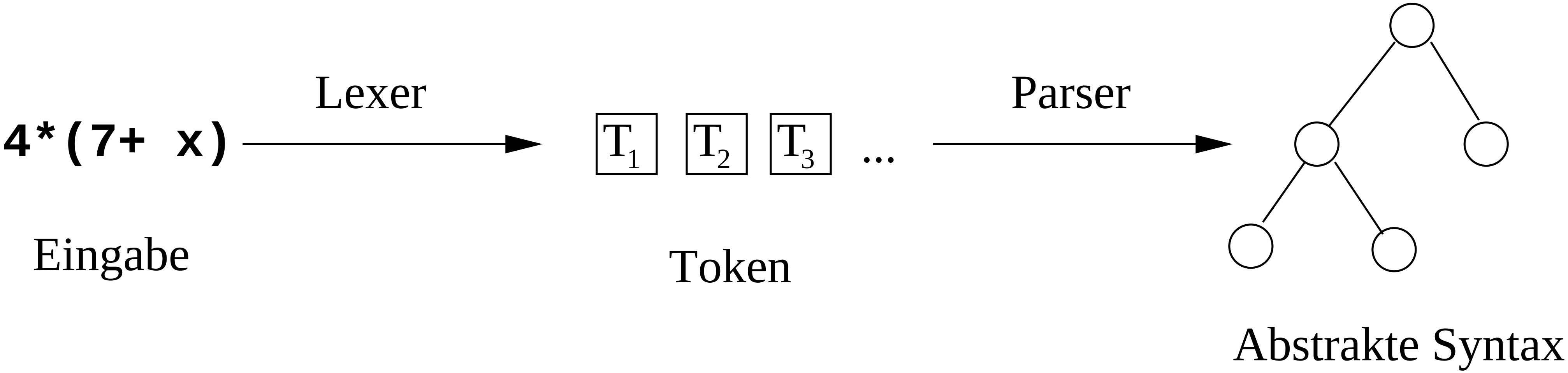
  - Zustandsbasierte Berechnungen in einem anderen Monaden `m`
  - `StateT s Id` ist Zustandsmonade
- Nicht sehr praktisch: alle Kombinationen müssen definiert werden.
  - Kombination von `State` und `Error`:

```
type ErrorState s a = Either String (s-> (s ,a))
type StateError s a = s-> (s, Either String a)
```

- Praktischer Ansatz: alles in den `IO`-Monaden.

Monadische Parserkombinatoren

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



- Parser bildet Eingabe auf Parsierungen ab.
- Basisparser erkennen Terminalsymbole
- Kombinatoren:
  - Sequenzierung (erst A, dann B)
  - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

$\Longrightarrow$ Nichtterminalsymbole

Arithmetische Ausdrücke

- Grammatik:

```
Expr   ::= Term + Term
         | Term - Term
         | Term
Term   ::= Factor * Factor
         | Factor / Factor
         | Factor
Factor ::= Number | (Expr)
Number ::= Digit  | Digit Number
Digit  ::= 0 |...| 9
```

- Daraus abstrakte Syntax:

```
data Expr    = Plus   Expr Expr
             | Minus Expr Expr
             | Times Expr Expr
             | Div    Expr Expr
             | Number Int
             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`
  - Müssen mehrdeutige Ergebnisse modellieren
  - Müssen Rest der Eingabe modellieren

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

- Beispiel: `parse "3+4*5"` $\rightsquigarrow$

```
[ (3, "+4*5"),
  Plus 3 4, "*5"),
  (Plus 3 (Times 4 5), "")]
```

Basisparser

- Erkennt nichts:

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

- Erkennt alles:

```
suceed :: b-> Parse a b
suceed 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 []
```

Kombinatoren

- 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]
```

- Eingabe weiterverarbeiten:

```
infix 4 `build`
build :: Parse a b-> (b-> c)-> Parse a c
build 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 `build` snd
p1 >* p2 = p1 >*> p2 `build` fst
```

Abgeleitete Kombinatoren

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

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

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

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

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

Der Kern des Parsers

- Parsierung von Expr

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm
                              `build` uncurry Plus
        `alt` pTerm >* token '-' >*> pTerm
                              `build` uncurry Minus
        `alt` pTerm
```

- Parsierung von `Term`

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pFactor
                              `build` uncurry Times
        `alt` pFactor >* token '/' >*> pFactor
                                `build` uncurry Div
        `alt` pFactor
```

- Parsierung von `Factor`

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

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen
- Zu prüfen: Testen.
  - Parsierung braucht Eingabe auf
  - 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 "Ambiguous input."
```

Ein kleiner Fehler

- Mangel: 3+4+5 ist Syntaxfehler
- Behebung: leichte Änderung der Grammatik . . .

```
Expr   ::= Term + Expr | Term - Expr | Term
Term   ::= Factor * Term | Factor / Term | Factor
Factor ::= Number | (Expr)
Number ::= Digit  | Digit Number
Digit  ::= 0 |...| 9
```

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

- Entsprechende Änderung des Parsers in pExpr

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr
                              ‘build‘ uncurry Plus
        ‘alt‘ pTerm >* token '-' >*> pExpr
                             ‘build‘ uncurry Minus
        ‘alt‘ pTerm
```

  - (vergleiche alt)

- . . . und in `pTerm`:

```
pTerm :: Parse Char Expr
pTerm = pFactor >* token '*' >*> pTerm
                           `build` uncurry Times
        `alt` pFactor >* token '/' >*> pTerm
                             `build` uncurry Div
        `alt` pFactor
```

- `pFactor` und Hauptfunktion bleiben: Testen.

Monadische Parserkombinatoren

- Der Parser ist ein Monade:

```
instance Monad (Parser a) where
  return a = \inp-> [(a, inp)]
  f >>= g  = \inp-> concat [g v out | (v, out)<- f inp]
```

- Explizite Parametrisierung über dieser Monade erlaubt Zustand beim Parsieren:
  - Symboltabellen
  - Zeilennummern

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Durch verzögerte Auswertung annehmbare Effizienz.
- Einfache Implementierung (wie oben) skaliert nicht
- Grammatik muß eindeutig sein (LL(1) o.ä.)
- Gut implementierte Büchereien (wie Parsec) bei eindeutiger Grammatik auch für große Eingaben geeignet

Zusammenfassung

- Monaden sind praktische Abstraktion
- Kombination bereitet Probleme, daher `IO` (die Mutter aller Monaden)
- Parserkombinatoren sind das Mittel der Wahl für flexible, performante Parser
  - `Parsec` mit dem GHC
- Ausblick: Nebenläufigkeit — Concurrent Haskell.