Vorlesung vom 09.12.2001: Verzögerte Auswertung und unendliche Datenstrukturen

Inhalt

- Verzögerte Auswertung
  - . . . und was wir davon haben.
- Unendliche Datenstrukturen
  - . . . und wozu sie nützlich sind.
- Fallbeispiel: Parserkombinatoren

Verzögerte Auswertung

- Auswertung: Reduktion von Gleichungen
  - Strategie: Von außen nach innen; von rechts nach links
- Effekt: Parameterübergabe *call by need*, nicht-strikt
  - Beispiel:

    `silly x y = y; double x = x+ x`

    `silly (double 3) (double 4)` $\leadsto$ `double 4` $\leadsto$ `4+ 4` $\leadsto 4$
  - Erstes Argument von `silly` wird nicht ausgewertet.
  - Zweites Argument von `silly` wird erst im Funktionsrumpf ausgewertet.

Striktheit

- Funktion `f` ist strikt in einem Argument `x`, wenn ein undefinierter Wert für `x` die Funktion undefiniert werden läßt.
- Beispiele:
  - `(+)` strikt in beiden Argumenten.
  - `(&&)` strikt im ersten, nicht-strikt im zweiten.
    `False && (1/0 == 1/0)` $\leadsto$ `False`
  - `silly` nicht-strikt im ersten. `silly (1/0) 3` $\leadsto$ `3`

Effekte der verzögerten Auswertung

- Datenorientierte Programme:
  - Berechung der Summe der Quadrate von 1 bis n:

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

    $\Longrightarrow$ Es werden keine Listen von Zwischenergebnissen erzeugt.
  - Minimum einer Liste durch

    ```
    min' xs = head (msort xs)
    ```

    $\Longrightarrow$ Liste wird nicht vollständig sortiert — binäre Suche.
- Unendliche Datenstrukturen.

Unendliche Datenstrukturen: Ströme

- Ströme sind unendliche Listen.
  - Unendliche Liste `[2,2,2,...]`

    ```
    twos = 2 : twos
    ```
  - Die 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. Zeigen.

    ```
    cycle (trace "Foo!" "x")
    ```
- Nützlich für Listen mit unbekannter Länge

Bsp: Berechnung der ersten n Primzahlen

- Eratosthenes — 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 :: Int-> [Integer]
primes n = take n (sieve [2..])
```

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.

- 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
```

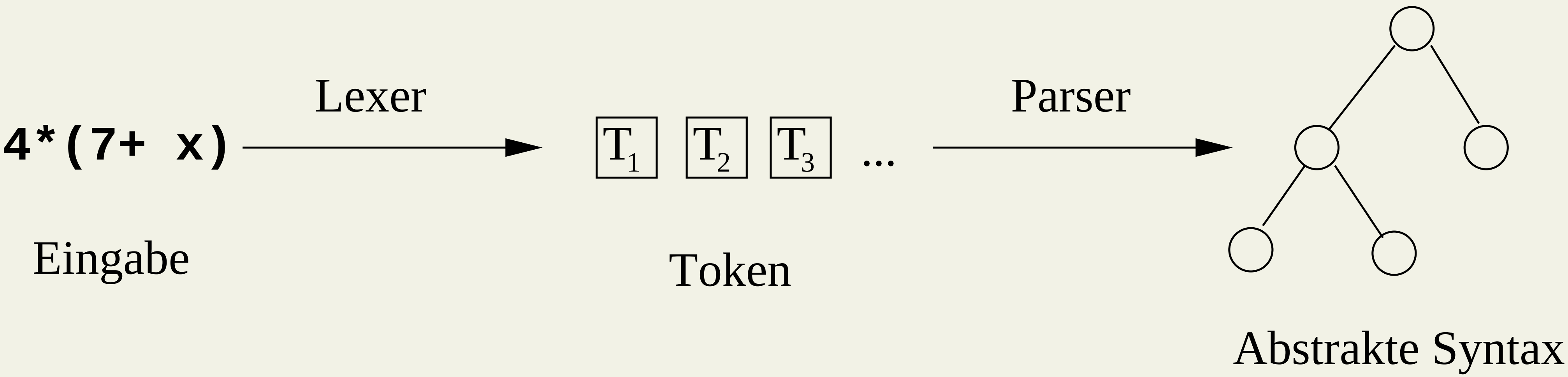
- Damit ergibt sich:

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

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

Fallstudie: Parsierung

- Gegeben: Grammatik
- Gesucht: Funktion, 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" ⇝ [ (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 []
```

  ◦ Warum nicht `none`, `suceed` durch `spot`?

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)

Einschub: Präzedenzen

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`
```

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.

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Abstraktion durch Funktionen höherer Ordnung.
- 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

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