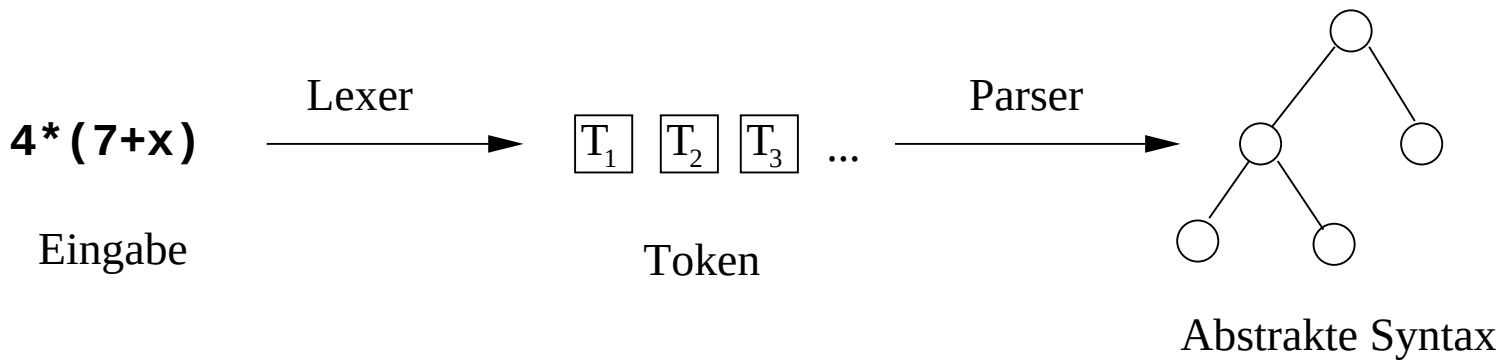
Praktische Informatik 3
Einführung in die Funktionale Programmierung
Parserkombinatoren

Christoph Lüth

WS 08/09

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

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term}\ \texttt{+}\ \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor}\ \texttt{*}\ \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Variable} \mid \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Variable} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{array}
$$

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?

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

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

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`?

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 =
  concatMap (\(b, r)->
     concatMap (\(c, s)-> [((b, c), s)]) (p2 r)). p1
```

Basiskombinatoren

- Ausgabe weiterverarbeiten:

```
infix 4 `use`, `use2`
use :: Parse a b-> (b-> c)-> Parse a c
use p f = map (\(o, r)-> (f o, r)) . p

use2 :: Parse a (b, c)-> (b-> c-> d)-> Parse a d
use2 p  = use p. uncurry
```

- 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 `use2` (:)
         `alt` suceed []
```

- 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)

Parsierung Arithmetischer Ausdrücke

- Token: `Char`
- Parsierung von `Expr`

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

- Parsierung von `Term`

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

Parsierung Arithmetischer Ausdrücke

- Parsierung von `Factor`

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

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

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term} + \mathit{Expr} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor} * \mathit{Term} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Variable} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \mathtt{a} \mid \cdots \mid \mathtt{z} \mid \mathtt{A} \mid \cdots \mid \mathtt{Z}
\end{array}
$$

- 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 Parserkombinatoren

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