Universität des Saarlandes
Fachrichtung Informatik
Programming Systems Lab
Prof. Gert Smolka
Proseminar Programmiersysteme WS 03/04

Höhere Funktionale Programmierung: Parser-Kombinatoren

Matthias Berg

Betreuer: Andreas Rossberg

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$
$$type\ Parser = String \rightarrow (String, Tree)$$

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$
$$type\ Parser = String \rightarrow (String, Tree)$$
$$type\ Parser\ result = String \rightarrow (String, result)$$

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$
$$type\ Parser = String \rightarrow (String, Tree)$$
$$type\ Parser\ result = String \rightarrow (String, result)$$
$$type\ Parser\ result = String \rightarrow [(String, result)]$$

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$
$$type\ Parser = String \rightarrow (String, Tree)$$
$$type\ Parser\ result = String \rightarrow (String, result)$$
$$type\ Parser\ result = String \rightarrow [(String, result)]$$
$$type\ Parser\ symbol\ result = [symbol] \rightarrow [([symbol], result)]$$

Definition Parser

Definition:
Ein Parser ist ein Algorithmus oder ein Programm, um die syntaktische Struktur einer Zeichenkette zu bestimmen.

Welchen Typ hat ein Parser?

$$type\ Parser = String \rightarrow Tree$$
$$type\ Parser = String \rightarrow (String, Tree)$$
$$type\ Parser\ result = String \rightarrow (String, result)$$
$$type\ Parser\ result = String \rightarrow [(String, result)]$$
$$type\ Parser\ symbol\ result = [symbol] \rightarrow [([symbol], result)]$$

$Eingabe$ $Reststring$ $Ergebnis$

Elementare Parser

Hier ein Parser, der ein bestimmtes Zeichen erkennt:

$$
\begin{array}{lll}
symbol & :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s & \\
symbol\ a\ [] & & = [] \\
symbol\ a\ (x : xs) & \mid\ a == x & = [(xs, x)] \\
& \mid\ otherwise & = []
\end{array}
$$

Elementare Parser

Hier ein Parser, der ein bestimmtes Zeichen erkennt:

$$
\begin{array}{lll}
symbol & :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s & \\
symbol\ a\ [] & & = [] \\
symbol\ a\ (x : xs) & \mid\ a == x & = [(xs, x)] \\
& \mid\ otherwise & = []
\end{array}
$$

Und ein Parser, der ein bestimmtes Wort erkennt:

$$
\begin{array}{lll}
token & :: Eq\ s \Rightarrow [s] \rightarrow\ Parser\ s\ [s] & \\
token\ k\ xs & \mid\ k == take\ n\ xs & = [(drop\ n\ xs, k)] \\
& \mid\ otherwise & = [] \\
& where\ n & = length\ k
\end{array}
$$

Parser Komposition

Sequentielle Komposition:

$$
\begin{aligned}
(<*>) \quad & :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ (a, b) \\
(p\ <*>\ q)xs = & [(xs2, (v1, v2)) \\
& |(xs1, v1) \leftarrow p\ xs \\
& , (xs2, v2) \leftarrow q\ xs1]
\end{aligned}
$$

"$p\ <*>\ q$" liefert einen Parser, der zuerst p auf die Eingabe anwendet und dann q auf den Reststring des Ergebnisses. Das Ergebnis ist das Paar aus den Ergebnissen der einzelnen Parser.

$$
\begin{aligned}
& > \ (token\ "ab"\ <*>\ symbol\ 'a')\ "abab" \\
& [("b", ("ab", 'a'))]
\end{aligned}
$$

Parser Komposition

Alternative Komposition:

$$(<|>) \qquad :: Parser\ s\ a \to Parser\ s\ a \to Parser\ s\ a$$
$$(p <|> q)xs = p\ xs +\!\!+ q\ xs$$

"$p <|> q$" liefert einen Parser, der p und q auf die Eingabe anwendet. Die Ergebnislisten werden konkateniert.

$$> \ (token\ "ab" <|> token\ "aba")\ "abab"$$
$$[("ab","ab"),("b","aba")]$$

Parser Transformation

$$
\begin{aligned}
sp \quad &:: Parser\ Char\ a \rightarrow Parser\ Char\ a \\
sp\ p \quad &= p\ .\ dropWhile(== \text{`}\ \text{`})
\end{aligned}
$$

$$
\begin{aligned}
just \quad &:: Parser\ s\ a \rightarrow Parser\ s\ a \\
just\ p &= filter(null\ .\ fst)\ .\ p
\end{aligned}
$$

"sp" verändert einen Parser derart, dass er führende Leerzeichen erlaubt.
"$just\ p$" erzeugt einen Parser, der seine Eingabe komplett verarbeiten muss.

Parser Transformation

$$
\begin{aligned}
(<@) \quad & :: Parser\ s\ a \to (a \to b) \to Parser\ s\ b \\
(p <@\ f)xs = & [(ys, f\ v) \\
& \quad |(ys, v) \leftarrow p\ xs]
\end{aligned}
$$

"$p <@\ f$" ist ein Parser, der wie p arbeitet, aber zusätzlich noch die Funktion f auf sein Ergebnis anwendet.

Parser Transformation

$$
\begin{aligned}
(<@) \quad &:: Parser\ s\ a \rightarrow (a \rightarrow b) \rightarrow Parser\ s\ b \\
(p <@\ f)xs &= [(ys, f\ v) \\
&\quad\ |(ys, v) \leftarrow p\ xs]
\end{aligned}
$$

"$p <@\ f$" ist ein Parser, der wie p arbeitet, aber zusätzlich noch die Funktion f auf sein Ergebnis anwendet.
Beispiel:

$$
\begin{aligned}
(<*) \quad &:: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a \\
p <*\ q &= p <*> q <@\ fst \\
(*>) \quad &:: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ b \\
p *>\ q &= p <*> q <@\ snd
\end{aligned}
$$

Sequentielle Komposition, die nur eines der beiden Ergebnisse berücksichtigt.

Parser Transformation

Anwendung:

$$
\begin{array}{ll}
(<*) & :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a \\
p <* q & = p <*> q <@ fst \\
(*>) & :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ b \\
p *> q & = p <*> q <@ snd
\end{array}
$$

$$
\begin{array}{ll}
succeed & :: r \rightarrow Parser\ s\ r \\
succeed\ v\ xs & = [(xs, v)]
\end{array}
$$

$$
data\ TREE = Nil \mid Bin(TREE,\ TREE)
$$

$$
\begin{array}{ll}
parens & :: Parser\ Char\ TREE \\
parens & = ((symbol\ '('\ *>\ parens <*\ symbol\ ')') <*> parens) \\
 & \qquad <@\ Bin \\
 & \quad <|> succeed\ Nil
\end{array}
$$

Noch mehr Kombinatoren

Hier ein sequentieller Kombinator, der Listenkonkatenation verwendet:

$$(<:*>) \quad :: Parser\ s\ a \to Parser\ s\ [a] \to Parser\ s\ [a]$$
$$p\ <:*>\ q = p\ <*>\ q\ <@\ (\lambda(x, xs) \to x : xs)$$

Noch mehr Kombinatoren

Hier ein sequentieller Kombinator, der Listenkonkatenation verwendet:

$$
\begin{aligned}
(<\!:\!*\!>) \quad &:: Parser\ s\ a \rightarrow Parser\ s\ [a] \rightarrow Parser\ s\ [a] \\
p\ <\!:\!*\!>\ q &= p\ <\!*\!>\ q\ <\!@\ (\lambda(x, xs) \rightarrow x : xs)
\end{aligned}
$$

Kleene-Stern Kombinator:

$$
\begin{aligned}
many \quad &:: Parser\ s\ a\ \rightarrow Parser\ s\ [a] \\
many\ p &= p\ <\!:\!*\!>\ many\ p \\
&\quad <|>\ succeed\ []
\end{aligned}
$$

Beispiele

$$
\begin{aligned}
&satisfy && :: (s \to Bool) \to Parser\ s\ s \\
&satisfy\ p\ [] && = [] \\
&satisfy\ p (x : xs) && = [(xs, x) \mid p\ x]
\end{aligned}
$$

$$
\begin{aligned}
&digit && :: Parser\ Char\ Int \\
&digit && = satisfy\ isDigit <@ (\lambda c \to (ord\ c) - (ord\ '0'))
\end{aligned}
$$

$$
\begin{aligned}
&natural && :: Parser\ Char\ Int \\
&natural && = many\ digit <@ foldl\ f\ 0 \\
& && \quad where\ f\ a\ b = a * 10\ +\ b
\end{aligned}
$$

$$
\begin{aligned}
&> \ natural\ "4711" \\
&[("", 4711), ("1", 471), ("11", 47), ("711", 4), ("4711", 0)]
\end{aligned}
$$

Beispiele

$$> \ natural\ "4711"$$
$$[("", 4711), ("1", 471), ("11", 47), ("711", 4), ("4711", 0)]$$

- natural ist gleichzeitig Lexer und Parser
- Laziness $\rightarrow$ Effizienz

Arithmetische Ausdrücke

$$TERM \quad ::= Nat \mid (\ TERM + TERM\ ) \mid (\ TERM - TERM\ )$$

$$\begin{aligned}
data\ Term\ &= \ Nat\ Int \\
&\mid \ Term :+: Term \\
&\mid \ Term :-: Term
\end{aligned}$$

$$\begin{aligned}
term\ &::\ Parser\ Char\ Term \\
term\ &=\ natural <@ Nat \\
&<|> open <*> term <*> plus <*> term <*> close \\
&\qquad <@ (\lambda(_, (a, (_, (b, _)))) \rightarrow a :+: b) \\
&<|> open <*> term <*> minus <*> term <*> close \\
&\qquad <@ (\lambda(_, (a, (_, (b, _)))) \rightarrow a :-: b) \\
&\quad where\ open = symbol\ '(' \quad close = symbol\ ')' \\
&\qquad plus = symbol\ '+' \quad minus = symbol\ '-'
\end{aligned}$$

Anwendungsbeispiel: BNF-Parser

Man kann eine Funktion definieren, die als Eingabe die Spezifikationen einer kontextfreien Grammatik bekommt und einen Parser zurückgibt, der die beschriebene Sprache erkennt.
Skizze:

$$data\ Tree \quad = Node\ String\ [Tree]$$

$$parsGen :: String \rightarrow Parser\ Char\ Tree$$
$$parsGen\ bnfstring\ = [...]$$

$$> \ parsGen\ "BLOCK ::= begin\ BLOCK\ end\ BLOCK \mid ."$$
$$"begin\ begin\ end\ begin\ end\ end"$$

Ausblick

- Fehlerkorrektur
- Fehlermeldungen
- Effizienzverbesserungen
- ...

Quellen

- Jeroen Fokker, Functional Parsers. Advanced Functional Programming, LNCS 925, Springer 1995
- Phil Wadler, How to Replace Failure by a List of Successes. Functional Programming Languages and Computer Architetcture, LNCS 201, Springer 1985
- Doaitse Swierstra, Combinator parsers: From toys to tools. Electronic Notes in Theoretical Computer Science 41, Elsevier Science Publisher 2001