Höhere Funktionale Programmierung: Parser-Kombinatoren

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

Matthias Berg
Betreuer: Andreas Rossberg

April 7, 2004

1 Einführung

In diesem Artikel werden einige Techniken vorgestellt, mit denen man in einer bedarfsgesteuerten (engl.: lazy), funktionalen Programmiersprache Parser programmieren kann. Die verwendete Programmiersprache ist Haskell. Die Implementierungen sollten sich aber recht einfach auf andere Sprachen übertragen lassen.

Das zweite Kapitel beschäftigt sich damit, welchen Typ Parser haben. Im dritten Kapitel werden elementare Parser für Zeichen und Wörter definiert. Kapitel 4 und 5 behandeln eine Reihe von Kombinatoren, mit denen sich Parser zu komplexeren Parsern verbinden lassen. Kapitel 6 und 7 realisieren als Anwendungsbeispiel Parser für geklammerte und arithmetische Ausdrücke. In Kapitel 8 wird ein Parser für Grammatiken entwickelt, der einen Parser für die Sprache der Grammatik erzeugt. So erhalten wir als mächtiges Werkzeug einen Parser Generator. Kapitel 9 beschäftigt sich mit Parsern, die in ihrer Eingabe Fehlerkorrekturen durchführen können.

2 Der Typ "Parser"

Wenn man vorhat einen Parser zu schreiben, ist eine der ersten Fragen, die man sich stellt "Welchen Typ hat denn ein Parser?". Als ersten Ansatz kann man sich überlegen, dass ein Parser einen String als Argument nimmt und daraus etwas Baumartiges berechnet, das die syntaktische Struktur dieses Strings beschreibt. Somit kommt man zu dem Typ

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

Wenn ein Parser die Möglichkeit haben soll, nicht die ganze Eingabe zu verarbeiten und nur ein Präfix der Eingabe zu betrachten, so ist es nützlich, den nicht verwendeten Teil der Eingabe wieder zurück zu geben, damit man diesen noch anderweitig verwenden kann. Zum Beispiel könnte man damit einen weiteren Parser füttern. Das ergibt folgenden Typ:

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

Es kann auch vorkommen, dass ein Parser mehrere Möglichkeiten hat, seine Eingabe zu parsen. Daher ist es hilfreich den Typ so anzupassen, dass ein Parser nicht nur ein einzelnes Ergebnis liefert, sondern gleich eine Liste von Ergebnissen:

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

Um den Parsertyp noch weiter zu verallgemeinern, wird nun der Typ der Eingabe und der Typ des Ergebnisses an *Parser* als Typargument übergeben:

```
type Parser symbol result = [symbol] → [ ( [symbol], result ) ]
```

Im Rest des Artikels wird diese Typdefinition verwendet. Die Symbolliste links des →-Zeichens wird nun als Eingabe, die Symbolliste rechts des →-Zeichens als Reststring und der *result*-Teil als Ergebnis bezeichnet. Mit Ergebnisliste ist die Liste von Paaren aus Reststrings und Ergebnissen gemeint.

3 Elementare Parser

Zunächst werden wir einen einfachen Parser definieren, der lediglich ein einzelnes Zeichen erkennt. Dazu schreiben wir eine Funktion *symbol*, die als Argument das Zeichen nimmt, das der resultierende Parser erkennen soll:

```
symbol              :: Eq s ⇒ s → Parser s s
symbol a []         = []
symbol a (x : xs) = [ (xs, x) | a == x ]
```

Der Parser überprüft, ob das erste Zeichen der Eingabe mit dem Argument a übereinstimmt. Wenn ja, so ergibt sich ein Paar aus nicht bearbeitetem Reststring xs und dem erkannten x.

Nun definieren wir einen Parser, der ein ganzes Wort erkennen kann. Erzeugt wird dieser Parser mit einer Funktion *token*, die als Argument das zu erkennende Wort bekommt:

```
token       :: Eq s ⇒ [s] → Parser s [s]
token k xs | k == take n xs = [(drop n xs, k)]
           | otherwise      = []
             where n        = length k
```

Und nun noch ein paar Parser, die später noch nützlich sein werden:

```
satisfy             :: (s → Bool) → Parser s s
satisfy p []        = []
satisfy p (x : xs) = [ (xs, x) | p x ]
```

satisfy bekommt als Argument ein Prädikat und ergibt einen Parser, der ein einzelnes Zeichen erkennt, für das dieses Prädikat zu *True* auswertet.

```
succeed       :: r → Parser s r
succeed v xs = [(xs, v)]
```

succeed erzeugt einen Parser, der das leere Wort akzeptiert. Das Ergebnis ist der Wert, der *succeed* übergeben wurde.

```
fails     :: Parser s r
fails xs = []
```

fails ist ein Parser, der alle Eingaben verwirft.

4 Parser Komposition

Man kann mehrere Parser miteinander kombinieren, um so komplexere Parser zu erhalten. Ein Beispiel hierfür ist die sequentielle Komposition. Wir definieren einen binären Infixoperator <∗>, der zwei Parser zu einem kombiniert, der sich so verhält, als ob die Parser einzeln "nacheinander" auf die Eingabe angewandt worden wären:

```
(<*>)          :: Parser s a → Parser s b → Parser s (a, b)
(p <*> q) xs = [ (xs2, (v1, v2))
               | (xs1, v1) ← p xs
               , (xs2, v2) ← q xs1 ]
```

Zunächst wird p auf die Eingabe angewandt. Der dabei entstehende Reststring wird dann an q übergeben. Das Ergebnis ist das Paar aus den Einzelergebnissen der Parser.

Hier Beispiel für die sequentielle Komposition:

```
> (token "ab" <*> symbol 'a') "abab"
[("b", ("ab", 'a'))]
```

An dem Reststring "b" erkennt man, dass der Parser das Wort "aba" erkennt.

Neben der sequentiellen gibt es noch die alternative Komposition. Hier werden die beiden Parser nicht "nacheinander", sondern "nebeneinander" auf die Eingabe angewandt. Das entspricht der Vereinigung der Ergebnisse der einzelnen Parser. Diese Vereinigung wird einfach durch Listenkonkatenation erreicht:

```
(<|>)          :: Parser s a → Parser s a → Parser s a
(p <|> q) xs = p xs ++ q xs
```

Hier ein Beispiel für die alternative Komposition:

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

Es gibt zwei Möglichkeiten die Eingabe zu parsen, wie man an der zweielementigen Liste sieht.

5 Parser Transformation

Es ist nützlich, sich ein paar Funktionen zu definieren, die Parser in einer bestimmten Art und Weise modifizieren können. Zum Beispiel kann man eine Funktion *sp* schreiben, die einen Parser derart verändert, dass er führende Leerzeichen erlaubt:

```
sp   :: Parser Char a → Parser Char a
sp p = p . dropWhile (== ' ')
```

Wie man sieht, geschieht dies durch Komposition mit einer Funktion, die von der Eingabe die führenden Leerzeichen einfach abschneidet.

Man kann auch eine Funktion *just* definieren, die einen Parser dazu bringt, nur noch Eingaben zu akzeptieren, die er komplett verarbeiten kann. Das bedeutet, dass der Reststring bei allen Ergebnissen leer sein muss.

```
just   :: Parser s a → Parser s a
just p = filter (null . fst) . p
```

Realisiert wird dies durch eine Funktion, die die Ergebnisliste nach leeren Reststrings filtert.

Nun kommen wir zu dem wichtigsten Transformator. Es handelt sich um einen Infixoperator, der einen Parser und eine Funktion als Argumente nimmt und einen Parser ergibt, der die Funktion auf sein Ergebnis anwendet.

```
(<@)           :: Parser s a → (a → b) → Parser s b
(p <@ f) xs = [ (ys, f v)
              | (ys, v) ← p xs ]
```

Ein Anwendungbeispiel des <@ Operator ist die Implementierung der folgenden zwei Operatoren:

```
(<*)  :: Parser s a → Parser s b → Parser s a
p <* q = p <*> q <@ fst

(*>)  :: Parser s a → Parser s b → Parser s b
p *> q = p <*> q <@ snd
```

Es handelt sich hierbei um Operatoren für sequentielle Komposition, wobei aber das Ergebnis nicht wie vorhin ein Paar der Einzelergebnisse, sondern nur eines der beiden Ergebnisse ist. Das Ergebnis eines Parsers wird hier einfach ignoriert. Erreicht wird dies durch die normale sequentielle Komposition, wobei mit dem <@ Operator noch die Funktion *fst* bzw. *snd* auf das Ergebnispaar angewandt wird. Solche Parser können nützlich sein, wenn man zwar möchte, dass ein Teil der Eingabe von einem Parser bearbeitet wird, man dessen Ergebnis aber nicht benötigt.

Ein weiterer sequentieller Kombinator, der mit Hilfe von <@ gebildet wird, ist der folgende:

```
(<:*>)  :: Parser s a → Parser s [a] → Parser s [a]
p <:*> q = p <*> q <@ (λ(x, xs) → x : xs)
```

Hier wird ein Parser erzeugt, der die Ergebnisse der einzelnen Parser konkateniert.

Sehr hilfreich ist auch folgender Transformator, der aus Parser p einen Parser erzeugt, der p beliebig oft sequentiell auf die Eingabe anwendet:

```
many  :: Parser s a → Parser s [a]
many p = p <:*> many p
         <|> succeed []
```

many realisiert den Kleene-Stern, der in regulären Ausdrücken verwendet wird.

Konfigurationsdateien haben oft ein Format, bei dem Werte mit irgendeinem Trennsymbol separiert werden. Um einen Parser für solch ein Format zu definieren, ist folgender Kombinator sehr nützlich:

```
listOf     :: Parser s a → Parser s b → Parser s [a]
listOf p s = p <:*> many (s *> p) <|> succeed []
```

listOf nimmt einen Parser p für die Werte und einen Parser s für die Separatoren als Argumente und erzeugt einen Parser, der als Ergebnis die Liste der geparsten Werte liefert. Mit Hilfe des *>-Operators, werden die nicht benötigten Ergebnisse von s entfernt.

Die folgenden zwei Funktionen nehmen jeweils eine Liste von Parsern als Argument und verknüpfen diese Parser dann sequentiell bzw. alternativ.

```
sequenceOf :: [Parser s a] → Parser s [a]
sequenceOf = foldr (<:*>) (succeed [])

choiceOf   :: [Parser s a] → Parser s a
choiceOf   = foldr (<|>) fails
```

6 Klammernausdrücke

Wir werden nun einen Parser für korrekt geklammerte Klammernausdrücke konstruieren. Immer wenn man etwas parsen will, muss man sich überlegen, was man als Ergebnistyp haben möchte. Bei Klammernausdrücken bieten sich natürlich Bäume an:

```
data TREE = Nil | Bin (TREE, TREE)

parens     :: Parser Char TREE
```

```
parens      = ((symbol '(' *> parens <* symbol ')') <*> parens) <@ Bin
              <|> succeed Nil
```

Hier werden die Operatoren $<*$ und $*>$ dazu verwendet, die Ergebnisse der *symbol*-Parser auszublenden. Das Ergebnis ist also nur das Paar der Ergebnisse der *parens*-Parser, auf die der Konstruktor *Bin* angewandt werden kann, um die Baumstruktur aufzubauen. *succeed* wird verwendet, um einen Rekursionsabbruch zu erreichen. *parens* erkennt also alle korrekt geklammerten Präfixe der Eingabe.

7 Arithmetische Ausdrücke

Nun werden wir einen Parser konstruieren, der natürliche Zahlen erkennt. Dafür benötigen wir zunächst einen Parser für Ziffern:

```
digit :: Parser Char Int
digit = satisfy isDigit <@ (λ c → (ord c) − (ord '0'))
```

Aus Digit lässt sich mit dem *many* Transformator leicht ein Parser für natürliche Zahlen konstruieren:

```
natural :: Parser Char Int
natural = many digit <@ foldl f 0
          where f a b = a * 10 + b
```

Da *many digit* beliebig oft auf die Eingabe anwendet, werden also alle möglichen Präfixe der Zahl erkannt, wie man an folgendem Beispiel sehen kann:

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

Wie man sieht, findet hier keine Unterscheidung zwischen Parser und Lexer mehr statt. Diese ist mit den vorgestellten Methoden nicht nötig.

Man könnte sich fragen, ob *natural* nicht sehr ineffizient ist, da ja bei jeder Zahl gleich auch alle Präfixe erkannt werden, von denen man in der Regel gar nichts wissen will. Hier kommt aber ins Spiel, dass Haskell eine bedarfsgesteuerte (engl.: lazy) Sprache ist und somit die Liste nur berechnet wird, wenn man sie auch braucht. Es ist wichtig zu erkennen, dass die Implementierung von *many* gierig (engl.: greedy) arbeitet. Das Ergebnis mit der komplett eingelesenen Zahl steht also am Anfang der Liste.

Man könnte *natural* aber auch statt mit *many* mit dem *greedy* Operator implementieren:

```
first       :: Parser a b  →Parser a b
first p xs  | null r       = []
            | otherwise    = [head r]
              where r      = p xs

greedy      = first . many
```

first verändert einen Parser derart, dass nur noch sein erstes Ergebnis in der Ergebnisliste auftaucht. Damit ist *greedy* ein Transformator, der einen Parser dazu bringt, sich möglichst oft sequentiell auf die Eingabe anzuwenden.

Ein etwas komplexeres Beispiel sind arithmetische Ausdrücke. Ein Term ist folgendermaßen definiert:

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

Wir definieren nun eine Datenstruktur *Term* um Terme zu repräsentieren und einen Parser *term*, der Terme erkennt:

```
data Term = Nat Int
          | Term :+: Term
          | Term :-: Term

term      :: Parser Char Term
term      = natural <@ Nat
          <|>symbol '(' <*> term <*> symbol '+' <*> term <*> symbol ')'
                <@ (λ(_,(a,(_,(b,_)))) → a :+: b)
          <|>symbol '(' <*> term <*> symbol '-' <*> term <*> symbol ')'
                <@ (λ(_,(a,(_,(b,_)))) → a :-: b)
```

Wie man sieht, gibt es eine enge Beziehung zwischen den Definitionen von *TERM*, *Term* und *term*. Strukturell ist ihr Aufbau sehr ähnlich und man kann sich denken, dass es recht einfach ist, aus der BNF-Definition einer Sprache einen Parser zu konstruieren, der die Sprache erkennt. Diese Beobachtung wird im nächsten Kapitel näher untersucht.

8 Parsergenerierung

Wir haben gesehen, dass man aus der Definition einer Sprache in BNF-Notation leicht einen zugehörigen Parser konstruieren kann. Nun stellt sich die Frage, ob man diese Konstruktion automatisieren kann. In diesem Kapitel werden wir eine Funktion definieren, die als Argument die Definition einer Sprache bekommt, und daraus einen Parser für die Sprache berechnet.

Zunächst müssen wir uns überlegen, wie man eine solche Sprachdefinition darstellen kann. In Grammatiken gibt es zwei verschiedene Arten von Symbolen: Terminalsymbole und Nichtterminalsymbole. Das führt uns zu folgendem Datentyp:

```
data Symbol = Term String | Nont String
```

Eine Grammatik ist eine Zuordnung von (Nichtterminal-) Symbolen zu einer Menge von Alternativen, die mit dem |-Zeichen voneinander getrennt werden. Eine Alternative ist wiederum eine Folge von Terminal- und Nichtterminalsymbolen. Eine solche (endliche) Zuordnung lässt sich durch eine Liste von Paaren repräsentieren:

```
type Alt  = [Symbol]
type Gram = [ (Symbol, [Alt]) ]
```

Um die Zuordnung auch als solche verwenden zu können, definieren wir folgende Funktion:

```
lookUp                 :: Eq s       ⇒[(s, d)] → s → d
lookUp ((u,v) : ws) x  | x == u      = v
                       | otherwise   = lookUp ws x
```

Wir werden nun einen Parser definieren, der einen BNF-String verarbeiten kann und als Ergebnis dessen Repräsentation von Typ *Gram* erzeugt. Damit der Parser zwischen Terminal- und Nichtterminalsymbolen unterscheiden kann, werden Hilfsparser für die beiden Symbolarten als Argument übergeben.

```
bnf                :: Parser Char String → Parser Char String → Parser Char Gram
bnf nontp termp    =many rule
    where rule     =( nont <*> sp (token " ::= ") *> rhs <* sp (symbol '.') )
          rhs      =listOf alt (sp (symbol '|'))
          alt      =many (term <|> nont)
```

```
term   =sp termp <@ Term
nont   =sp nontp <@ Nont
```

Nun folgt ein Beispiel, dass die Benutzung von *bnf* deutlich macht. Als Parser für Terminal- bzw. Nichtterminalsymbole werden Parser verwendet, die einfach nur Großbuchstaben- bzw. Kleinbuchstabenwörter erkennen:

```
> bnf upper lower "BLOCK ::= begin BLOCK end BLOCK | ."

[("", [(Nont BLOCK, [ [Term "begin"
                      , Nont "BLOCK"
                      , Term "end"
                      , Nont "BLOCK"], []])]),
 ("BLOCK ::= begin BLOCK end BLOCK | .", [])]
```

Um von Grammatiken erzeugte Sprachen parsen zu können, benötigt man eine allgemeine Datenstruktur, die all diese Sprachen darstellen kann. Es liegt auf der Hand, dass man hierfür eine Baumstruktur verwenden sollte, die die verwendeten Produktionsregeln der Grammatik repräsentiert:

```
data Tree = Node Symbol [Tree]
```

Die Blätter eines solchen Baums entsprechen den Symbolen der zu erkennenden Eingabe. Innere Knoten tragen immer Nichtterminalsymbole, wobei jedes dieser Symbol zusammen mit den Symbolen der Kinder des Knotens einer Regelalternative in der Grammatik entspricht.

Die folgende Funktion erzeugt aus einer Grammatik und einem zugehörigem Startsymbol einen Parser, der als Ergebnis einen solchen Baum liefert:

```
parsGram                    :: Gram → Symbol → Parser Symbol Tree
parsGram gram start         =parsSym start
    where parsSym           :: Symbol → Parser Symbol Tree
          parsSym s@(Term t) =(symbol s <@ const []) <@ Node s
          parsSym s@(Nont n) =parsRhs (lookUp gram s) <@ Node s
          parsAlt           :: Alt → Parser Symbol [Tree]
          parsAlt           =sequenceOf . map parsSym
          parsRhs           :: [Alt] → Parser Symbol [Tree]
          parsRhs           =choiceOf . map parsAlt
```

bnf und *parsGram* werden nun so kombiniert, dass aus der BNF-Definition einer Sprache ein Parser für diese Sprache auf Symbol-Ebene berechnet wird:

```
symParsGen :: Parser Char String → Parser Char String → String → Symbol →
              Parser Symbol Tree
symParsGen nontp termp bnfstring start =
              (snd . head . (bnf nontp termp <@ parsGram)) bnfstring start
```

Der so entstandene Parser erwartet als Eingabe eine Liste von Symbolen. Unser Ziel ist aber ein Parser, der direkt einen String einlesen kann. Dazu benötigt man einen Lexer, der die Eingabe in die einzelnen Symbole zerlegt. Eines der Argumente von *symParsGram* ist der Parser für Terminalsymbole *termp*. Dieser ist im Prinzip schon ein Lexer für die Sprache. Die folgende Funktion nutzt dies aus:

```
parsGen :: Parser Char String → Parser Char String → String → Symbol →
           Parser Char Tree
parsGen nontp termp bnfstring start xs =
           [(rest, tree)
```

```
    |(rest, tokens) ← many ((sp termp) <@ Term) xs
    ,(_, tree) ← just (symParsGen nontp termp bnfstring start) tokens]
```

Mit *parsGen* haben wir nun also einen Parsergenerator für kontextfreie Grammatiken definiert.

Das folgende Beispiel realisiert einen Parser für das einfach getypte λ-Kalkül:

```
> (snd . head)
  (parsGen
    upper
    (lower <|> choiceOf (map token ["->", " : ", "(", ")", "_"]))
    " TYP ::= bool | int | real | TYP -> TYP .
      TERM ::= ( TERM ) | X | lam X : TYP _ TERM | TERM TERM .
      X ::= x | y | z."
    (Nont "TERM")
    "lam y : int _ lam x : int->real _ ( x y )"
  )

Node (Nont "TERM") [
    Node (Term "lam") [],
    Node (Nont "X") [
        Node (Term "y") []
    ],
    Node (Term " : ") [],
    Node (Nont "TYP") [
        Node (Term "int") []
    ],
    Node (Term "_") [],
    Node (Nont "TERM") [
        Node (Term "lam") [],
        Node (Nont "X") [
            Node (Term "x") []
        ],
        Node (Term " : ") [],
        Node (Nont "TYP") [
            Node (Nont "TYP") [
                Node (Term "int") []
            ],
            Node (Term "->") [],
            Node (Nont "TYP") [
                Node (Term "real") []
            ]
        ],
        Node (Term "_") [],
        Node (Nont "TERM") [
            Node (Term "(") [],
            Node (Nont "TERM") [
                Node (Nont "TERM") [
                    Node (Nont "X") [
                        Node (Term "x") []
                    ]
                ],
                Node (Nont "TERM") [
                    Node (Nont "X") [
                        Node (Term "y") []
```

```
                        ]
                    ]
                ],
                Node (Term ")") []
            ]
        ]
    ]
```

9 Fehlerkorrektur

Die bisherige Parserdefinition bringt ein paar Nachteile mit sich. Wenn z.B. die Eingabe aufgrund von Tippfehlern nicht geparst werden kann, so bekommt man als Ergebnis nur eine leere Liste und keinen Hinweis darauf, an welcher Stelle das Parsen fehlschlug. Es kann sogar passieren, dass bei dem Versuch doch eine Lösung zu finden, ein Stack Overflow auftritt. Für größere Eingaben, die z.B. von Hand geschrieben wurden und daher oft Tippfehler enthalten, kann dies sehr unpraktisch werden.

Wir werden nun einen neuen Ansatz verwenden, der es uns ermöglicht, Parser zu schreiben, die Fehlerkorrekturen durchführen. So können z.B. kleinere Tippfehler automatisch verbessert werden. Die Parser werden ihre Eingabe im Gegensatz zu den bisherigen Parsern mit einer Breitensuche durchmustern. Ermöglicht wird dies durch Continuation basiertes Programmieren. So lassen sich schon während des Parsens Informationen über den bisher erreichten Fortschritt sammeln. Diese kann man durch Ausnutzen der "laziness" von Haskell dazu verwenden, nicht benötigte Fälle frühzeitig zu eliminieren.

Um zu verdeutlichen, was Continuation basiertes Programmieren für unsere Parser bedeutet, kommt hier ein Beispiel, das soweit vereinfacht ist, dass als Ergebnis je nach Zugehörigkeit zur Sprache nur *True* oder *False* geliefert werden:

```
type Parser symbol = ([symbol] → Bool) → ([symbol] → Bool)

succeed              :: Parser s
succeed r input      = r input

symbol               :: Eq s ⇒ s → Parser s
symbol a r []        = False
symbol a r (b : bs)  = a == b && r bs

(<*>)                :: Parser s → Parser s → Parser s
(p <*> q) r input    = p (q r) input

(<|>)                :: Parser s → Parser s → Parser s
(p <|> q) r input    = p r input || q r input

parse                :: Parser s → [s] → Bool
parse p input        = p null input
```

Die Funktion *parse* kann man verwenden um einen Parser auf eine Eingabe anzuwenden. Die hier als Continuation verwendete Funktion *null* prüft am Ende nach, ob die Eingabe komplett verarbeitet wurde. Generell enthält der Continuationparameter die Information darüber, was noch zu tun ist, wenn der Parser seine Arbeit beendet hat.

Ein wesentlicher Schritt Richtung Fehlerkorrektur ist die Einführung einer Datenstruktur, die es uns ermöglicht, den Fortschritt des Parsevorganges zu verfolgen. Wir unterscheiden zwischen zwei verschiedenen Arten von Parseschritten:

Ok-Schritte bezeichnen die erfolgreiche Erkennung eines Symbols

Fail-Schritte bezeichnen eine Fehlerkorrektur: Entweder das Einfügen oder das Löschen eines Symbols in der Eingabe.

```
data Steps result = Ok   (Steps result)
                  | Fail (Steps result)
                  | Stop result
```

Ein vollständiger Parsevorgang besteht also aus einer Folge von *Ok*- und *Fail*-Schritten. Optimal wäre es also zu einer Eingabe das Ergebnis zu finden, das die wenigsten *Fail*-Schritte benötigt. Dies ist aber kaum möglich, ohne dass man im wesentlichen alle Korrekturmöglichkeiten betrachtet. Davon kann es aber exponentiell viele geben. Der folgende Algorithmus wählt aus zwei Schrittfolgen mit Hilfe einer greedy-Strategie die (hoffentlich) Bessere aus. Als "besser" wird hier die Folge mit dem längeren Präfix von *Ok*-Schritten definiert.

```
best :: Steps rslt  →Steps rslt  →Steps rslt
best   (Ok l)       (Ok r)       = Ok (best l r)
best   (Fail l)     (Fail r)     = Fail (best l r)
best   l@(Ok _)     (Fail _)     = l
best   (Fail _)     r@(Ok _)     = r
best   l@(Stop _)   _            = l
best   _            r@(Stop _)   = r
```

Man beachte, dass die "laziness" von Haskell hier bewirkt, dass die "schlechtere" Folge nur so weit wie nötig berechnet wird.

Nun können wir Parser schreiben, die Fehlerkorrekturen durchführen und mit Hilfe von *best* sinnvoll zwischen den Korrekturen wählen.

```
type Parser symbol = ([symbol] → Steps Bool) → ([symbol] →Steps Bool)

symbol              :: Eq s ⇒ s → Parser s
symbol a r []       = Fail (r [])                          {− Insert symbol a −}
symbol a r (b : bs) = if a == b then Ok(r bs)
                      else Fail(best(r (b : bs))           {− Insert symbol a −}
                                    (symbol a r bs))       {− Delete symbol b −}

(<|>)               :: Parser s → Parser s → Parser s
(p <|> q) r input   = best (p r input) (q r input)

(<∗>)               :: Parser s → Parser s → Parser s
(p <∗> q) r input   = p (q r) input

parse               :: Parser s → [s] → Steps Bool
parse p input       = p (foldr (const Fail) (Stop True)) input
```

Der <|>-Operator und die *symbol* Funktion werden mit Hilfe von *best* definiert. Dadurch wird die Eingabe mit einer Breitensuche durchmustert. *parse* verwendet als Continuation nun eine Funktion, die eine Folge von *Fail*-Schritten erzeugt, deren Länge mit der Länge der restlichen Eingabe übereinstimmt. Dies entspricht einer wiederholten Löschung, bis die Eingabe leer ist.

Die so konstruierten Parser führen zwar eine Fehlerkorrektur durch, haben aber als Ergebnis stets den Wert *True*. Wir werden die Parserdefinitionen nun so erweitern, dass auch ein Ergebnis berechnet wird. Das schon berechnete Ergebnis werden wir in einem zusätzlichen Argument anhäufen. Damit erhalten wir folgenden Parsertyp:

```
type Future sym result = [result] → [sym] → Steps [result]
```

```
type Parser sym result = Future sym result          {- Continuation -}
                       → [result]                   {- bisheriges Ergebnis -}
                       → [sym]                      {- Eingabe -}
                       → Steps [result]
```

An den Definitionen von <∗> und <|> ändert sich kaum noch etwas. Interessant ist nun vor allem die Implementierung von *symbol*. Die Fehlerkorrektur arbeitet wie im vorherigen Beispiel. Das berechnete Ergebnis wird nun als Argument an die Continuation weitergegeben. Wie man sieht, ist *token* nichts anderes als eine Folge von *symbol*-Parsern.

```
(<∗>)                       :: Parser s r → Parser s r → Parser s r
(p <∗> q) c st inp          = p (q c) st inp

(<|>)                       :: Parser s r → Parser s r → Parser s r
(p <|> q) c st inp          = best (p c st inp) (q c st inp)

symbol                      :: Eq s ⇒ s → Parser s s
symbol a                    = parser
where parser c st (b : bs)  = if a == b then Ok(c (st++[a]) bs)
                              else Fail(best (parser c st bs)
                                             (c (st++[a]) (b : bs)))
      parser c st []        = Fail(c (st++[a]) [])

epsilon                     :: Parser s r
epsilon c st inp            = c st inp

sequenceOf                  :: [Parser s r] → Parser s r
sequenceOf                  = foldr (<∗>) epsilon

token                       :: Eq s ⇒ [s] → Parser s s
token                       = sequenceOf . (map symbol)

parse                       :: Parser s r → [s] → Steps [r]
parse p xs                  = p (λ st inp → foldr (const Fail) (Stop st) inp) [] xs
```

Als Anwendungsbeispiel wählen wir wieder das einfach getypte λ-Kalkül:

```
xpars     =  symbol 'x' <|> symbol 'y' <|> symbol 'z'

typpars   =  token "unit"
          <|>(symbol '(' <∗> typpars <∗> token "->" <∗> typpars <∗> symbol ')')

termpars  =  xpars
          <|> (token "lam" <∗> xpars <∗> symbol ':' <∗> typpars <∗> symbol '.'
               <∗> termpars)
          <|>(symbol '(' <∗> termpars <∗> symbol ' ' <∗> termpars <∗> symbol ')')
```

So lassen sich nun λ-Terme erkennen, in denen auch kleinere Tippfehler vorkommen:

```
> parse termpars "lam x : unit.x"
Ok (Ok (Ok (Ok (Ok (Ok (Ok (Ok (Ok (Ok (Ok (Ok
(Stop "lam x : unit.x")))))))))))

> parse termpars "lma x : unt..x"
Ok (Fail (Ok (Fail (Ok (Ok (Ok (Ok (Ok (Fail (Ok (Ok (Fail (Ok
```

```
(Stop "lam x : unit.x")))))))))))))))
```

Es gibt auch Beispiele, bei denen der Vorgang der Fehlerkorrektur nicht so leicht nachzuvollziehen ist:

```
> parse termpars "hello"
Fail (Fail (Ok (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Ok (Fail (Fail (Fail (Fail
(Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail
(Stop "lam x : unit.lam x : unit.x")))))))))))))))))))))))))))

> parse termpars "(l : ())"
Ok (Ok (Fail (Fail (Fail (Fail (Ok (Ok (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail (Fail
(Fail (Ok (Fail (Fail (Fail (Fail (Ok
(Stop "(lam x : (unit->unit).x x)")))))))))))))))))))))))
```

Um den Vorgang der Fehlerkorrektur etwas leichter nachvollziehbar zu machen, kann man neben dem Argument, in dem die Ergebnisse angehäuft werden, ein weiteres Argument verwenden, in dem Fehlermeldungen gesammelt werden. So ist es am Ende leichter zu verstehen, was für Korrekturen vorgenommen wurden.

```
type Future sym result      = [result] → [String] → [sym] → Steps ([String], [result])

type Parser sym result      = Future sym result      {- Continuation -}
                            → [result]               {- bisheriges Ergebnis -}
                            → [String]               {- Fehlermeldungen -}
                            → [sym]                  {- Eingabe -}
                            → Steps ([String], [result])

symbol :: (Eq s, Show s)    ⇒ s → Parser s s
symbol a                    = parser
where parser c st err (b : bs) = if a == b then Ok(c (st++[a]) err bs)
                                 else Fail(best(parser c st (err++["Deleted "++ show b])bs)
                                           (c (st++[a]) (err++["Inserted "++ show a])(b : bs)))
      parser c st err []    = Fail(c (st++[a]) (err++["Inserted "++ show a])[])

getresult                   :: Steps result → result
getresult (Ok l)            = getresult l
getresult (Fail l)          = getresult l
getresult (Stop v)          = v

parse :: Show s             ⇒ Parser s r → [s] → ([String], [r])
parse p xs                  = getresult (p (λ st err inp → foldr (const Fail)
                              (Stop (err++["Deleted "++ show i| i ← inp], st)) inp) [] [] xs)
```

Die Implementierungen von <*>, <|>, *epsilon* und *token* ändern sich nur unwesentlich und wurden weggelassen. Nun sehen die Beispiele von oben folgendermaßen aus:

```
> parse termpars "hello"
(["Deleted 'h'","Deleted 'e'","Inserted 'a'","Inserted 'm'","Inserted ' ','Inserted 'x'",
  "Inserted ' :' ","Inserted 'u'","Inserted 'n'","Inserted 'i'","Inserted 't'","Inserted '.'",
  "Deleted 'o'","Inserted 'a'","Inserted 'm'","Inserted ' ','Inserted 'x'","Inserted ' :' ",
  "Inserted 'u'","Inserted 'n'","Inserted 'i'","Inserted 't'","Inserted '.'","Inserted 'x'"],
  "lam x : unit.lam x : unit.x")
```

```
> parse termpars "(l : ())"
(["Inserted 'a'","Inserted 'm'","Inserted ' '","Inserted 'x'","Inserted 'u'","Inserted 'n'",
  "Inserted 'i'","Inserted 't'","Inserted '-'","Inserted '>'","Inserted 'u'","Inserted 'n'",
  "Inserted 'i'","Inserted 't'","Inserted '.'","Inserted 'x'","Inserted ' '","Inserted 'x'"],
  "(lam x : (unit->unit).x x)")
```

10 Zusammenfassung

Funktionales Programmieren ermöglicht Implementierungen von Parsern, die sich auf elegante Art und Weise mittels sogenannter Parser-Kombinatoren miteinander verknüpfen lassen. Es gibt Kombinatoren, die Parser sequentiell oder alternativ verknüpfen, und solche, die die Eigenschaften eines Parsers verändern können (Transformatoren). So lassen sich recht einfach Parser für komplexe Sprachen definieren. Wir haben auch einen Parser geschrieben, der als Eingabe die Definition einer Sprache bekommt, und daraus einen Parser für diese Sprache berechnet.

Darüber hinaus wurden Techniken vorgestellt, mit denen man Fehler korrigierende Parser konstruieren kann. So ist es auch möglich Eingaben zu erkennen, die nicht genau der Sprachdefinition entsprechen.

Es gibt Techniken, mit denen sich diese Parser noch verbessern lassen. In Swierstras Paper [3] wird u.a. gezeigt, wie sich deren Effizienz noch erhöhen lässt.

11 Quellen

1. Jeroen Fokker, Functional Parsers. Advanced Functional Programming, LNCS 925, Springer 1995

2. Phil Wadler, How to Replace Failure by a List of Successes. Functional Programming Languages and Computer Architetcture, LNCS 201, Springer 1985

3. Doaitse Swierstra, Combinator parsers: From toys to tools. Electronic Notes in Theoretical Computer Science 41, Elsevier Science Publisher 2001