Monadic Parsing in Haskell

Eine Ausarbeitung zum gleichnamigen Paper von G. Hutton u. E. Meijer

Lukas Convent

Universität des Saarlandes

Zusammenfassung

Diese Ausarbeitung widmet sich dem Thema Parsing in Haskell, wobei zwei verschiedene Herangehensweisen - *Applicative Parsing* und *Monadic Parsing* - genauer untersucht und gegenübergestellt werden. Bei der Präsentation von bestimmten Parserkonstrukten wird der Fokus auf die sich eröffnenden Möglichkeiten durch funktionale Programmierung unter Verwendung von Monaden gelegt. Auf die Betrachtung, wie der Lexing-Prozess mit Hilfe der vorgestellten Möglichkeiten integriert werden kann, folgt noch ein Ausblick auf weiterführende Aspekte sowie ein abschließendes Beispiel.

1 Einführung

Die Konstruktion von Parsern gehört seit jeher zu einer wichtigen Aufgabe, die in verschiedenen Umgebungen immer wieder durchgeführt werden muss. Sehr effiziente Parser erhält man heutzutage durch maschinelle Generierung von Bottom-Up-Parsern, welche in den meisten Fällen den hier betrachteten Parsern in puncto Geschwindigkeit überlegen sind. Anstatt auf Effizienz wird der Fokus in dieser Ausarbeitung auf Eleganz und Erweiterbarkeit gelegt. Das Ziel sind Parser, aus denen man intuitiv die geparste Sprache ableiten kann und welche sich einfach anhand einer gegebenen Grammatik konstruieren lassen. Hierzu wird auf die Modularität zurückgegriffen, die uns in der funktionalen Programmierung zum Beispiel durch Monaden zur Verfügung steht.

Verzichtet man auf die später eingeführten Konzepte, so tritt häufig das Problem auf, Grammatiken tauglich zu machen für den *Rekursiven Abstieg*. *RA-Tauglichkeit* besagt hierbei Folgendes:

1. In jeder Rekursion verkürzt sich die Argumentliste.
2. Anhand des ersten Wortes lässt sich feststellen, welche Produktionsregel verwendet wird.

Offensichtlich gibt es viele Grammatiken, welche diesen Bedingungen nicht genügen. Eines der prominentesten Beispiele ist das folgende:

```
exp  ::=  [exp "+"] mexp
mexp ::=  ...
```

Durch die Linksassoziativität des +-Operators erhält man eine linksrekursive Grammatik, welche Bedingung 2 von *RA-Tauglichkeit* nicht erfüllt. Um dennoch zum Ziel zu kommen, kann Folgendes getan werden:

1. Die Grammatik in eine rechtsrekursive umwandeln. Dafür nimmt man zunächst in Kauf, dass der +-Operator rechtsassoziativ wird:

   ```
   exp  ::=  mexp ["+" exp]
   ```

2. In den konkreten Parser-Prozeduren setzt man sich nun über die Grammatik hinweg und fügt den Additions-Baum (mittels Übergabe von bereits geparstem Baum) linksassoziativ zusammen.

Nun wäre es schön, wenn man von diesem Standardvorgehen auch im Programm abstrahieren könnte: Auf diese Weise blieben einem das Umschreiben der Grammatik sowie die Anpassungen im Parser erspart. Dies soll als Motivation genügen, denn für genau diese Fälle sollen elegante Lösungen präsentiert werden.

2 Parser-Objekte

Zunächst legen wir fest, mit welchen Parser-Objekten überhaupt gearbeitet wird. Hierzu führen wir folgenden Typen `Parser` ein, zusammen mit einer Hilfsprozedur zum Zugriff auf den Kern des Parsers:

```
newtype Parser a = Parser (String -> [(a, String)])

parse (Parser f) = f
```

Der Kern des Parsers besteht aus einer Prozedur, welche einen Eingabestring auf eine Liste von Ergebnissen abbildet. Jedes Ergebnis enthält neben dem geparsten Ausdruck immer auch den Reststring des Parse-Vorgangs. Die Rückgabe der leeren Liste steht also für einen erfolglosen Parse-Durchgang, bei mehrdeutigen Grammatiken können aber auch entsprechend mehrere Ergebnisse auftreten.

Die Flexibilität durch Listen kommt auch dem Parsen von Sprachen mittels Backtracking zugute: Hier können zunächst mehrere alternative Zwischenergebnisse entstehen, die dann genutzt werden können, wenn zuerst gewählte Pfade sich als unergiebig herausstellen.

Nun können wir unseren ersten Parser definieren, `item`. Dieser parst genau ein `char` und liefert ihn als Ergebnis:

```
item  :: Parser Char
item  =  Parser (\cs -> case cs of
                          ""   -> []
                          c:cr -> [(c, cr)])
```

Mittels der `parse`-Prozedur können wir den Parser nun anwenden und erhalten das gewünschte Ergebnis:

```
parse item     "haskell"     ⇝     [('h', "askell")]
```

Die Idee ist jetzt, komplexere Parser aus einfachen zusammenzusetzen. Um diesen Prozess des Zusammenfügens zu demonstrieren, stellen wir uns folgende Aufgabe: Wir möchten einen Parser `charPair :: Parser (Char, Char)` erschaffen, welcher die ersten beiden Zeichen liest und sie als Paar zurückgibt:

```
parse charPair "haskell"     ⇝     [(('h', 'a'), "skell")]
```

Es gibt jetzt mehrere Möglichkeiten, den schon bestehenden Parser `item` so zu verknüpfen, dass wir den Parser `charPair` erhalten.

3 Parser sequenzialisieren

Um Parser auf elegante Weise zu verknüpfen, kann man ihnen eine Struktur zuweisen. Wie schon der Titel andeutet, interpretieren wir den `Parser`-Datentypen als Monade und implementieren die entsprechenden Prozeduren. Die monadische Struktur (`Monad`) impliziert immer auch die

applikative Struktur (`Applicative`)[1]. Im Folgenden werden wir deshalb zunächst den Parser als `Applicative` betrachten, und anschließend die monadische Struktur „oben aufsetzen“.

3.1 Applicative

Die Typklasse `Applicative` fordert zwei Prozeduren, die nun für `Parser` implementiert werden:

```
instance Applicative Parser where
   pure                        :: a -> Parser a
   pure x                      =  Parser (\cs -> [(x, cs)])

   (<*>)                       :: Parser (a -> b) -> Parser a -> Parser b
   (Parser fp) <*> (Parser p) =  Parser (\cs -> [(f a, cs'') |
                                    (f, cs') <- fp cs, (a, cs'') <- p cs'])
```

`pure`. Die Prozedur nimmt einen Wert entgegen und liefert einen Parser, in dessen Ergebnisraum bereits jener Wert liegt. Kommt es zu einer Ausführung des Parsers, so bleibt der Eingabestring erhalten und wird zusammen mit dem injizierten Wert als alleiniges Ergebnis zurückgeliefert. Als *Lifting*-Prozedur entspricht sie genau dem `return` der Typklasse `Monad`.

`(<*>)`. Der Sequenzoperator erzeugt einen Parser, welcher zwei zuvor übergebene Parser hintereinander ausführt. Hierzu wird bei einer Ausführung zunächst der erste Parser ausgeführt, der Ergebnisse (mit begleitenden Reststrings) vom Typ `a -> b` liefert. Der zweite Parser wird nun auf jeden dieser zugehörigen Reststrings angewandt, was in einer Liste von Ergebnissen des Typs `a` resultiert. Nun wird jede geparste Prozedur auf seine entsprechenden Argumente angewandt: Der Typ der Ergebnisse ist somit jeweils `b`.

Wir können nun den gewünschten Parser `charPair` auf folgende Weise realisieren:

```
charPair :: Parser (Char, Char)
charPair =  pure (\c d -> (c, d)) <*> item <*> item
```

Wie man erkennt, ist bereits vor der Ausführung der beiden `item`-Parser determiniert, wie die Ergebnisse weiterverarbeitet werden. Dies muss nicht notwendigerweise der Fall sein, wie die weitere Möglichkeit der Sequenzialisierung über den `>>=`-Operator der Monade zeigt.

3.2 Monad

Während die *Lifting*-Prozedur `return` genau dieselbe Funktion wie `pure` erfüllt, stattet der `>>=`-Operator `Parser` mit einer neuen Variante der Parser-Verknüpfung aus:

```
instance Monad Parser where
   return    :: a -> Parser a
   return x  =  pure x

   (>>=)     :: Parser a -> (a -> Parser b) -> Parser b
   p >>= f   =  Parser (\cs -> concat [ parse (f a) cs' | (a, cs') <- parse p cs ])
```

[1] Aus historischen Gründen wird diese Eigenschaft erst seit GHC 7.10 explizit gefordert (*Functor-Applicative-Monad Proposal*)

`(>>=).` Der *bind*-Operator erzeugt einen Parser, welcher zunächst den ersten Parser ausführt. Anhand der Ergebnisse dieses Parsers wird mittels der übergebenen Prozedur ein zweiter Parser bestimmt, welcher dann die Endergebnisse liefert.

Eine zweite Möglichkeit, `charPair` zu definieren, ist die nun folgende:

```
charPair :: Parser (Char, Char)
charPair =  do
               c <- item
               d <- item
               return (c, d)
```

Der Unterschied von `>>=` zu `<*>` tritt hierbei hervor: Dadurch, dass man sich erst nach Ausführen des ersten Parsers auf den nachfolgenden Parser festlegt, gewinnt man an Flexibilität. Über die übergebene Prozedur kann man dem Folgeparser eine Art „Kontext“ mitgeben. Für die weiteren vorgestellten Parser wird gemäß dem Titel ein *monadic style* anstelle eines *applicative style* verfolgt. Eine Gegenüberstellung der beiden Stile befindet sich in Kapitel 7.

4 Parser wählen lassen

In jeder nichttrivialen Grammatik gibt es Fälle, in denen mehrere Produktionsregeln verfolgt werden können. Um eine Auswahl zwischen zwei Parsern (welche mit zwei Produktionsregeln korrespondieren) zu ermöglichen, führen wir den nichtdeterministischen Auswahloperator `mplus` ein.

Dazu gehört ein `mzero`-Element, welches einen ergebnislosen Parser darstellt. Die Kombination von `mplus` und `merzo` ermöglicht den Durchlauf verzweigter Parsingbäume mit Backtracking. Zusammen instanziieren sie die Typklasse `MonadPlus` für `Parser`[2]:

```
instance MonadPlus Parser where
   mzero        :: Parser a
   mzero        =  Parser (\cs -> [])

   mplus        :: Parser a -> Parser a -> Parser a
   p `mplus` q  =  Parser (\cs -> parse p cs ++ parse q cs)
```

Wegen der Konkatenation der beiden Ergebnislisten in `mplus` ist sichergestellt, dass beide Möglichkeiten vollständig berücksichtigt werden; dank *lazy evaluation* werden die Parsingpfade jedoch nur auf Anforderung berechnet. Für die meisten Zwecke reicht es aus, einen einzigen erfolgreichen Parsingpfad zu betrachten, weshalb ein deterministischer Auswahloperator `(+++)` definiert wird:

```
(+++)   :: Parser a -> Parser a -> Parser a
p +++ q =  Parser (\cs -> case parse (p `choice` q) cs of
                            [] -> []
                            (x:xs) -> [x])
```

Für die weiteren vorgestellten Parser kann tatsächlich `(+++)` verwendet werden. Eine detaillierte Gegenüberstellung von `mplus` und `(+++)` folgt in Kapitel 8.

[2]Im referierten Paper wird auf `(++)` statt `mplus` sowie `zero` statt `mzero` verwiesen, welche den beiden veralteten Typklassen `MonadZero` und `MonadPlus` entspringen

5 Grundbausteine

Die Grundbausteine stellen einige elementare Parser sowie weitere sogenannte **Parserkombinatoren** dar. Ein Parserkombinator ist eine Prozedur, welche Parser als Argumente entgegennimmt und einen maßgeschneiderten Parser zurückliefert. Ausgestattet mit der Fähigkeit zum Sequenzialisieren und Auswählen, können jetzt einige sehr nützliche Parser sowie Parserkombinatoren konstruiert werden. Hierbei steht die folgende Auswahl nur stellvertretend für die Fülle an Möglichkeiten, die sich anbietet:

`sat`. Anstatt ein beliebiges Zeichen zu parsen, liefert `sat` nur ein Zeichen als Ergebnis, wenn es ein übergebenes Prädikat erfüllt:

```
sat    :: (Char -> Bool) -> Parser Char
sat p  =  do
   c <- item
   if p c then return c else mzero
```

`char`. Neben Parsern für Ziffern, Großbuchstaben etc. kann aus `sat` ein Parser geschaffen werden, der ein spezifisches Zeichen parst:

```
char    :: Char -> Parser Char
char c  =  sat (c ==)
```

`string`. Mit `string` führen wir den ersten rekursiven Parser ein, welcher durch seinen Parameter die Länge der Rekursionsfolge beschränkt:

```
string    :: String -> Parser String
string s  =  case s of
   ""   -> return ""
   c:cr -> do
               char c
               string cr
               return (c:cr)
```

`many`. Bei `many` handelt es sich um einen Parserkombinator, welcher ein häufiges Parsingmuster implementiert: Die beliebig häufige Ausführung eines Parsers. Somit entspricht er der *Kleene'schen Hülle* z.B. bei den regulären Ausdrücken. Dieser Parserkombinator verfolgt wegen der Verwendung von `(+++)` das *maximal munch*-Prinzip: Weil der „gierige“ Operand an erster Stelle steht, wird das einzige Ergebnis jenes mit den meisten Parserausführungen sein. Eine Variante `many1` mit mindestens einer Parserausführung lässt sich ganz ähnlich formulieren.

```
many   :: Parser a -> Parser [a]
many p =  (do { e <- p; es <- many p; return (e:es) })
          +++
          (return [])
```

`sepby`. Um eine Folge von Ausdrücken zu parsen, von denen zwei Elemente jeweils durch einen speziellen Trennausdruck voneinander getrennt sind, kann ein Parserkombinator `sepby` definiert werden. Eine Besonderheit ist hier, dass die Trennausdrücke ignoriert und nur die geparsten Elemente als Liste zurückgeliefert werden. Die Variante `sepby1` für nichtleere Folgen wird analog definiert.

```
sepby          :: Parser a -> Parser b -> Parser [a]
p `sepby` sep  =  (do
                     e <- p
                     es <- many (do { sep; p })
                     return (e:es))
                  +++
                  (return [])
```

5.1 Linksrekursive Grammatiken

Wie in der Einführung schon vorgestellt, erschweren linksrekursive Grammatiken die Konstruktion von Parsern. Wir betrachten im Folgenden den Fall, dass eine Kette von Ausdrücken geparst werden soll, welche durch einen linksassoziativen Operator verbunden sind, z.B.

```
1 + 2*2 + 3*3 + (4+1)     ⇝     ((1 + 2*2) + 3*3) + (4+1)
```

Die korrespondierende (linksrekursive) Grammatik ist dieselbe wie in der Einführung:

```
exp  ::=  [exp "+"] mexp
mexp ::=  ...
```

Der Parserkombinator `chainl1` stellt nun eine universelle Lösung für diese Aufgabe dar: Gegeben ein Parser für die einzelnen Ausdrücke (z.B. für *mexp*) sowie ein Parser für den Operator (z.B. für $+$), kapselt `chainl1` den Parsingprozess inklusive der Linksrekursion in einem Parser:

```
chainl1         :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op  =  do { a <- p; rest a }
   where
      rest a = (do { f <- op; b <- p; rest (f a b) } )
               +++
               (return a)
```

Die *interne* rekursive Prozedur `rest` bekommt als Argument immer einen bereits geparsten Ausdruck übergeben und faltet dank *maximal munch*-Prinzips die längstmögliche Kette von Ausdrücken zu einem Ausdruck zusammen. Ob es sich bei einem Ausdruck nun um eine *abstract syntax tree*-Ausprägung oder ein direkt evaluiertes Ergebnis handelt, ist dem Anwender des Parserkombinators überlassen. Eine entsprechende Variante `chainr1` für rechtsassoziative Operatoren lässt sich ähnlich formulieren.

6 Lexing

Bisher unerwähnt blieb der Lexing-Prozess, welcher typischerweise dem Parsing vorangeht. Die eingeführten Parser operieren direkt auf Strings statt auf *tokens*, was einen einfachen Grund hat: Die bis hierhin geschaffenen Konstrukte können auch für das Lexing genutzt werden. Wir führen hierbei den Lexing-Prozess nicht separat aus, sondern integrieren ihn mittels Parserkombination:

- Um eine der Hauptfunktionen des Lexings zu erfüllen, der Abgrenzung von *tokens* durch *white space*, kann `space` als Hilfsparser bemüht werden:

  ```
  space  :: Parser String
  space  =  many (sat isSpace)
  ```

- Der Parserkombinator `token` führt einen Parser aus und wirft anschließendes *white space* weg.

  ```
  token    :: Parser a -> Parser a
  token p  =  do { a <- p; space; return a }
  ```

Weitere lexikalische Parser, z.B. zum Entfernen von Kommentaren in Quelltexten oder das Parsen von Zahlen, Schlüsselwörtern können je nach Bedarf ähnlich definiert werden.

7 *Applicative style* vs. *Monadic style*

Wie eingangs schon erwähnt, wird einem mit dem >>=-Operator ein mächtiges Werkzeug in die Hand gegeben. Im Folgenden soll betrachtet werden, zu welchem Zweck der unflexiblere *applicative style* statt des *monadic style* verwendet werden kann. Anschließend wird auf die Mächtigkeit der beiden Parsing-Paradigmen eingegangen.

7.1 Statische Analyse mittels *applicative style*

Die eingeschränkte Flexibilität des Parsens via <*> birgt nicht nur Nachteile: Die Tatsache, dass die sequenzialisierten Parserobjekte statisch festgelegt sind (im Gegensatz zur dynamischen Determinierung bei >>=), ermöglicht eine statische Analyse der Parser. So kann man Informationen über sequenzialisierte Parser erhalten, wie z.B.:

- Liefert der Parser für einen leeren Eingabestring Ergebnisse?
- Was ist die Menge der ersten Zeichen, die ein Parser konsumieren kann?

Zunächst muss eine Möglichkeit geschaffen werden, diese Informationen für elementare Parser zur Verfügung zu stellen. Anschließend kann z.B. für eine Sequenzialisierung von zwei Parsern festgehalten werden, dass der sequenzialisierte Parser genau dann für einen leeren Eingabestring Ergebnisse liefert, wenn beide Parser dies tun.

Um diese statische Analyse wirklich umzusetzen, müssen noch einige Änderungen vorgenommen werden. Anregungen liefert hierbei [5] und eine Anwendung findet sich in folgender Arbeit [4], welche zeigt, wie statische Analyse zur Fehlerbehebung genutzt werden kann.

7.2 Mächtigkeit

Nun stellt sich die Frage, ob man mit dem *monadic style*-Parsing mehr Sprachen parsen kann als im *applicative style*. Im *applicative style* kann jede kontextfreie Sprache geparst werden: Für jede syntaktische Kategorie wird ein Parser konstruiert, welcher genau die jeweils möglichen Produktionsregeln anwendet.

Dadurch, dass der Folgeparser durch den >>=-Operator anhand des Ergebnisses des ersten Parsers und somit abhängig von einem beliebig definierbaren Kontext erstellt wird, können im *monadic style* auch kontextsensitive Sprachen geparst werden. Betrachten wir eines der Paradebeispiele für kontextsensitive Sprachen, die Sprache $a^n b^n c^n$. Ein monadischer Parser kann nun zunächst die a^n-Sequenz parsen und als Ergebnis die Länge der Sequenz, n, als Kontext an zwei Folgeparser übergeben. Diese parsen eine b- bzw. c-Sequenz der mitgelieferten Länge n.

Es stellt sich heraus ([6]), dass die Sprache $a^n b^n c^n$ auch problemlos im *applicative style* geparst werden kann. Der Grund ist, dass wir durch Rekursion implizit eine unendlich lange Grammatik erzeugen können:

```
        S ::= SeqA(0)

SeqA(0) ::= "a" SeqA(1) | SeqB(0) SeqC(0)
SeqA(1) ::= "a" SeqA(2) | SeqB(1) SeqC(1)
SeqA(2) ::= ...
```

```
  ⋮

SeqB(0) ::= ϵ
SeqB(1) ::= "b" SeqB(0)
SeqB(2) ::= "b" SeqB(1)

  ⋮

SeqC(0) ::= ϵ

  ⋮
```

Die Parameter der Nichtterminale `SeqA(n)`, `SeqB(n)` und `SeqC(n)` speichern hierbei die Information, wie viele `"a"`-Terminale bereits gelesen wurden bzw. noch gelesen werden müssen. Diese Grammatik ist nun wegen ihrer unbeschränkten Größe nicht mehr kontextfrei, kann aber unmittelbar in rekursive, applikative Parserprozeduren übersetzt werden.

8 Determinismus vs. Nichtdeterminismus

In den bis jetzt vorgestellten Parsern und Kombinatoren war die Entscheidung, ob der deterministische oder der nichtdeterministische Auswahloperator verwendet werden sollte, immer zugunsten des deterministischen gefallen. Im Folgenden soll auf die Vorteile von Determinismus eingegangen werden sowie auf die Frage, wann er Nichtdeterminismus ersetzen kann.

8.1 Vorteile von Determinismus

Der Hauptvorteil des deterministischen Auswahloperators - manchmal wegen seiner „Links-Präferenz“ auch *biased choice operator* genannt - ist das Verhindern von unnötigem Speicherverbrauch. Auch wenn dank *lazy evaluation* die nicht benötigten Pfade nicht ausgewertet werden, so werden sie doch im Speicher abgelegt als potentiell zu evaluierende Pfade (jeweils mit Referenz auf den Eingabestring). Es kann häufig sehr viele Pfade geben, wenn man bedenkt, dass es bei jedem `space`-Aufruf pro geparstem Leerzeichen zwei Abzweigungen gibt. Wann immer sein Einsatz möglich ist, verspricht der `(+++)`-Operator durch Abschneiden nicht mehr benötigter Pfade also ein Potential an Speichereinsparung.

Ein weiterer Vorteil zeigt sich bei der Laufzeit, wenn bei einer ungültigen Eingabe Backtracking angewendet wird, obwohl dies aussichtslos ist. Dies kann mitunter bei der „verzweifelten“ Suche nach einem Ergebnis dazu führen, dass der ganze potentielle Parsingbaum unnötigerweise abgelaufen wird.

8.2 Verwendung von Determinismus

Dass der deterministische Auswahloperator in den bisherigen Parsern und Kombinatoren verwendet wurde, hängt mit dem *maximal munch*-Prinzip zusammen. Dieses kam in allen bisherigen Anwendungen, z.B. `sepby` zum Tragen. Nur der längstmögliche geparste Ausdruck führt letztendlich zum Ziel. Würde z.B. beim Parsen einer Folge `"1,2,3"` nur die Folge `"1,2"` geparst, so verbliebe ein Reststring `",3"`, von dem angenommen wird, dass er für Folgeparser ungültig ist.

Diese Annahme muss natürlich nicht stimmen. Betrachten wir hierfür ein minimales Beispiel: Es soll die Sprache $\{ab,\ abb\}$ geparst werden. Ein erster Ansatz wäre folgender:

```
p  :: Parser ()
p  =  do { (string "ab" +++ string "a"); string "b"; return () }
```

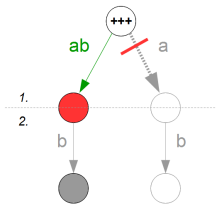

Abbildung 1: *"ab"* kann nicht geparst werden

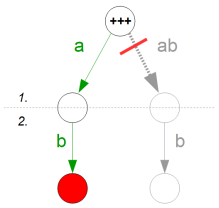

Abbildung 2: *"abb"* kann nicht geparst werden

Der Parser scheitert jedoch beim Parsen des Wortes `"ab"` (s. Abbildung 1). Der `(+++)`-Operator präferiert den linken Parser, welcher erfolgreich `"ab"` konsumiert. Nun wird der rechte Parsingpfad verworfen und der Parser steckt in einer Sackgasse, weil kein `"b"` mehr gelesen werden kann.

Auch das Austauschen der beiden Operanden lässt vergeblich auf einen korrekten Parser hoffen:

```
p  =  do { (string "a" +++ string "ab"); string "b"; return () }
```

Hier ist das Parsen von `"abb"` nicht mehr möglich, wenn abschließend alle Ergebnisse als ungültig betrachtet werden, deren Reststrings nicht leer sind (s. Abbildung 2). Es bestehen nun zwei Ansätze, das Problem zu lösen:

Nichtdeterministischer Auswahloperator. Das Ersetzen von `(+++)` durch `mplus` sorgt dafür, dass die Parsingpfade sich bis zur nächsten nichtdeterministischen Entscheidung behaupten müssen (in diesem Fall bis zum Ende). Dies entspricht für eindeutige Grammatiken also einer Erweiterung des *lookahead* um eine Ebene, wobei unter Ebene eine Parser-Sequenz bis zur nächsten nichtdeterministischen Entscheidung (oder dem Ende) zu verstehen ist. Um eine vorschnelle Festlegung auf einen möglicherweise falschen Pfad zu vermeiden, wird eine weitere Ebene „vorausgeschaut“ und damit sichergestellt, dass ein Scheitern des Parsingvorgangs aufgrund einer falschen Entscheidung in diesem Punkt nicht stattfindet (s. Abbildung 3):

```
p  =  do { (string "a" `mplus` string "ab"); string "b"; return () }
```

Faktorisierung In diesem Beispiel kann durch Umstellen des Parsers der gemeinsame Präfix `"ab"` „herausfaktorisiert“ werden. Auf diese Weise muss `"ab"` nicht für jeden Pfad neu geparst werden. Gleichzeitig kann der Folgeparser `return ()` wegen der Linksdistributivität von `(+++)`[3] in die beiden Operanden „multipliziert“ werden. Da sich der Entscheidungspunkt nun auf letzter Ebene befindet, reicht der *lookahead* über eine Ebene aus und es kann unter Verwendung des *maximal munch*-Prinzips `(+++)` verwendet werden (s. Abbildung 4):

```
p  =  do { string "ab"; do { string "b"; return ()} +++ return ()}
```

9 Beispiel und Ausblick

Ein abschließendes Beispiel soll nun demonstrieren, dass wir unserem ursprünglichen Ziel, eine Methode zur eleganten Übersetzung von Grammatiken in modulare Parser zu finden, sehr nahe

[3] Die Linksdistributivität von `mplus` aus `MonadPlus` überträgt sich auf `(+++)`

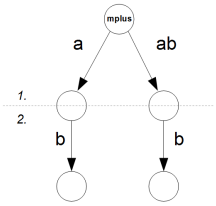

Abbildung 3: Verwendung von *mplus*

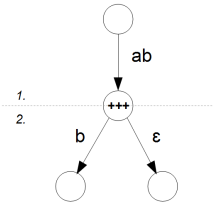

Abbildung 4: Faktorisierter Parser

gekommen sind. Hierfür betrachte man einen Parser `exp` für eine einfache Teilmenge der arithmetischen Ausdrücke; die korrespondierende Grammatik ist in den Kommentaren beigefügt:

```
data Exp = Con Int | Id Char | Sum Exp Exp | Pro Exp Exp

exp  :: Parser Exp
exp  = chainl1 mexp plus                        -- exp   ::= [exp "+"] mexp
mexp = chainl1 pexp times                       -- mexp  ::= [mexp "*"] pexp
pexp = digit +++ id +++ (do { symb "(";         -- pexp  ::= digit | id | "(" exp ")"
                              e <- exp;
                              symb ")";
                              return e })

plus  = do { symb "+"; return Sum }
times = do { symb "*"; return Pro }
digit = do { x <- token (sat isDigit);          -- digit ::= 0 | ... | 9
             return (Con (ord x - ord '0')) }
id    = do { name <- token (sat isAlpha);       -- id    ::= a | ... | z | A | ... | Z
             return (Id name) }
```

Das hier vorgestellte Framework kann mit weiteren Verbesserungen angereichert werden, z.B. einer Fehlerdiagnose oder sogar automatischer Fehlerkorrektur (welche nicht beim ersten Fehler abbricht, sondern so viel wie möglich parst). Hierfür können *monad transformers* verwendet werden, welche das Verschachteln von Monaden ermöglichen. Im Gegensatz zur *Happy*-Bibliothek, die eine automatische Generierung von *bottom-up*-Parsern anbietet, bauen Bibliotheken wie *Parsec* und *Attoparsec* auf den in dieser Ausarbeitung erklärten Verfahren auf und verallgemeinern sie noch etwas. Sowohl *applicative style* als auch *monadic style* stehen einem hier zur Verfügung und laden zum Gebrauch eleganter funktionaler Konzepte ein.

Literatur

[1] Graham Hutton, Erik Meijer *Monadic parsing in Haskell* 1998.

[2] Daan Leijen, Erik Meijer *Parsec: Direct Style Monadic Parser Combinators for The Real World* 2001.

[3] Daan Leijen *Parsec, a fast combinator parser* 1998.

[4] `https://stackoverflow.com/questions/7861903/what-are-the-benefits-of-applicative-parsing-ove 7863380#7863380`