

Parsen in der funktionalen Programmiersprache Haskell

Autor: Jens Kulenkamp

10. Dezember 2003

Diese Dokumentation ist im Rahmen des Informatik-Seminar entstanden.

Student, Matrikel: Jens Kulenkamp, [ii4799@fh-wedel.de]
Fachbereich: Technische Informatik
Betreuer: Prof. Dr. Schmidt
Abgabetermin: Dezember 2003

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1. Ein kurzer Überblick	4
2. Grundlagen	5
2.1. Parser	5
2.2. Grammatik	5
2.2.1. mehrdeutige Grammatik	5
2.2.2. eindeutige Grammatik	6
2.2.3. Linksrekursion	7
3. Funktionale Parser in Haskell	8
3.1. Motivation	8
3.2. Der Datentyp Parse	8
3.3. Elementar Parser	9
3.3.1. symbol	9
3.3.2. spot	10
3.3.3. dig	10
3.3.4. succeed	10
3.4. Parser kombinatoren	11
3.4.1. Die Alternative	11
3.4.2. Die Sequenzialisierung	12
3.4.3. Der Umwandler	12
3.4.4. many	13
3.4.5. option	14
3.5. Ein Parser für arithmetische Ausdrücke	14
3.5.1. Die Idee	14
3.5.2. Die Grammatik	14
3.5.3. Der Parser	15

4. Parsec	17
4.1. Was ist Parsec?	17
4.2. Beispiele mit Parsec	17
4.2.1. Die Sequenz und Auswahl	18
4.2.2. Parser mit Semantik	18
4.2.3. Die Auswahl	19
4.2.4. Ein Taschenrechner mit Parsec	20
5. Download	22
Literaturverzeichnis	23

1. Ein kurzer Überblick

Diese Ausarbeitung ist im Rahmen des Informatik-Seminar im Wintersemester 2003/2004 entstanden.

Es werden im folgenden Kapitel zunächst ein paar Grundlagen abgehandelt. Hierbei wird erklärt, was ein Parser ist und auf die Definition von Grammatiken eingegangen. Damit die Grammatikdefinition etwas Verständlicher wird, folgen zwei Beispiele, welche anschließend diskutiert und bewertet werden. Im Anschluß dieser Beispiele folgt die vollständige Entwicklung eines nicht deterministischen Parser in der funktionalen Programmiersprache Haskell. Zunächst wird hierfür ein allgemeiner und wiederverwendbarer Datentyp entwickelt. Auf Grundlage dieses Typs werden zunächst elementar Parser (*engl. elementary Parser*) und später Parser kombinatoren (*engl. Parser combinator*) entwickelt. Im Anschlußdieser Entwicklungsphase wird auf Grundlage der erarbeiteten Parser ein Taschenrechner für arithmetische Ausdrücke entwickelt.

2. Grundlagen

2.1. Parser

Ein Parser ist ein Programm, das einen Text oder Token als Eingabe erhält. Diese Eingabe wird syntaktisch analysiert und in eine logische Struktur umgewandelt. Der Text kann zum Beispiel eine Programmiersprache, eine XML-Datei aber auch ein arithmetischer Ausdruck oder Datenbankbefehl sein. Die Ausgabe eines Parsers ist normalerweise ein Syntaxbaum oder Programmbaum. Dieser Syntaxbaum wird anhand der gegebenen Grammatik erstellt, wobei diese den Sprachumfang bzw. die Ableitungsregeln der zu analysierenden Sprache umfassen muss. Entspricht die Eingabe nicht der gegebenen Syntax, so sind entsprechende Fehlermeldungen zu generieren.

2.2. Grammatik

Für die Entwicklung eines Parsers ist zunächst die Definition einer Grammatik notwendig, damit der Parser den einzulesenden Text erkennen und strukturieren kann. Eine kontextfreie Grammatik ist ein Quadrupel $G = (T, N, P, S)$ wobei gilt:

T : ist ein Alphabet von Terminalsymbolen.

N : ist ein Alphabet von nichtterminalen Symbolen und beschreiben strukturierte Einheiten.

P : ist eine Menge von Produktionsregeln. Eine Produktionsregel besteht aus einem Paar von Nichtterminal und einer Folge von Nichtterminal- und/oder Terminalsymbolen. Produktionsregeln sind auch die Ersetzungsregeln.

S : ist das Startsymbol. Die Menge aller aus dem Startsymbol S ableitbaren Terminalsymbolen T ist die erzeugte Sprache G .

2.2.1. mehrdeutige Grammatik

Im folgenden ist eine Grammatik G_1 für arithmetische Ausdrücke gegeben.

$$G_1 = (T, N, P, S)$$

$$T = \{num, +, -, *, /\}$$

$$N = \{E\}$$

$$P = \{E\}$$

$$E = \{E \rightarrow num | E + E | E * E | E - E | E / E\}$$

$$S = E$$

Hier steht E für *expression*.

Als Beispiel wird folgender Ausdruck betrachtet $10+2$.

Der Parser wird diesen Ausdruck von links nach rechts lesen, wobei ausgehend vom Startsymbol, E sich wie folgt ableiten lässt: $E \rightarrow E + E \rightarrow num + E \rightarrow num + num$. Anhand dieser Ableitungsfolge lässt sich der zugehörige Syntaxbaum erstellen und wie folgt darstellen.

Syntaxbaum 1

Der dargestellte Syntaxbaum ist zunächst ausgehend von der Wurzel von oben nach unten aufgebaut. Dieses Prinzip nennt sich *Topdown Analyse*. Desweiteren lässt sich feststellen, dass sich an den Knoten die nichtterminalen Symbole und an den Blättern die terminalen Symbole befinden.

Als weiteres Beispiel betrachtet man das Terminalwort $4*5-8$ und erstellt ebenfalls die Ableitungsfolge und den dazugehörigen Syntaxbaum.

$$E \rightarrow E * E \rightarrow num * E \rightarrow num * E - E \rightarrow num * num - E \rightarrow num * num - num$$

$$E \rightarrow E - E \rightarrow E - num \rightarrow E * E - num \rightarrow num * E - num \rightarrow num * num - num$$

Syntaxbaum 2 3

Hierbei lässt sich feststellen, dass sich für den gleichen Ausdruck mehrere Syntaxbäume erstellen lassen. Diese Syntaxbäume entstehen durch Anwendung unterschiedlicher Produktionsregeln. Das Problem hierbei liegt in der mehrdeutigen Semantik. Es darf nicht sein, dass ein Ausdruck unterschiedliche Bedeutung haben kann. Das Ergebnis muss durch die Grammatik eindeutig definiert sein. Mehrdeutigkeiten sind bei der Entwicklung einer Grammatik zu vermeiden. Ein möglicher Lösungsansatz ist zum Beispiel die Festlegung von Prioritäten, Assoziativitäten oder eine natürlich strukturierte Grammatik zum Beispiel durch Klammerung.

2.2.2. eindeutige Grammatik

Betrachtet man die gegebene Grammatik G_2 .

$$G_2 = (T, N, P, S)$$

$$T = \{num, +, -, *, /\}$$

$$N = \{E, D, F\}$$

$$P = \{E \rightarrow E + D \mid E - D \mid D, D \rightarrow D * F \mid D / F \mid F, F \rightarrow num\}$$

$$S = E$$

F steht für *factor*. Durch das zusätzliche Symbol F ist eine links Assoziativität definiert und eine Mehrdeutigkeit ausgeschlossen.

Wird für den gleichen Ausdruck $4*5-8$ der dazugehörige Syntaxbaum betrachtet, so ist zu erkennen, dass vor der Berechnung des Ausdrucks $E - D$ der Kindknoten $D * F$ berechnet werden muss. Desweiteren ist festzuhalten, dass für den Ausdruck nur noch ein Syntaxbaum erstellt werden kann.

Syntaxbaum 4

Durch die Regel $D \rightarrow D * F \mid D / F \mid F$ ist die links Assoziativität festgelegt worden. Tauscht man zunächst den Ausdruck durch folgende Regel aus $D \rightarrow F * D \mid F / D \mid F$ so lässt sich eine rechts Assoziativität implementieren.

2.2.3. Linksrekursion

Bei der gegebenen Grammatik G_2 aus dem Abschnitt 2.2.2 kann die Produktion der Form $A \rightarrow A\alpha$ zu Problemen führen. Gemeint ist hierbei, dass das Nichtterminalsymbol A auf der rechten Seite der Produktion wieder ganz links auftritt (Rekursion). Betrachtet man einen Ausschnitt der bekannten Regeln $E \rightarrow E + D | D$, und überlegt sich eine mögliche Implementierung, so könnte diese wie folgt aussehen:

```
void E() {
    E();
    Symbol(); // Symbol + ueberlesen!
    T();
}
```

Bei der Betrachtung der gegebenen Implementierung wird sich $E()$ rekursiv aufrufen welches unwiederbringlich zu einem Stacküberlauf führen wird. Dieser Implementierungsfehler lässt sich jedoch durch folgende Änderung der Grammatik beheben $E \rightarrow D + E | D$. Durch diese Änderung wurde aus einer Linksrekursion eine Rechtsrekursion.

3. Funktionale Parser in Haskell

3.1. Motivation

In diesem Kapitel werden unterschiedliche elementar Parser und Parser kombinatoren entwickelt. Für die Kommunikation der einzelnen Parser ist es jedoch zwingend erforderlich, einen Datentyp `Parse` zu entwickeln. Dieser Datentyp soll möglichst allgemein und wiederverwendbar sein, so dass es keine Rolle spielt, ob zum Beispiel Strings oder Tokens verarbeitet werden. Um dieses Ziel zu erreichen, wird der Datentyp vom allgemeinsten Fall zunächst schrittweise verfeinert und abstrahiert.

3.2. Der Datentyp `Parse`

Stellt man sich zunächst einen vereinfachten Parser als Blackbox vor und betrachtet lediglich die Ein- und Ausgabe eines Parsers, so wird ein Parser einen String als Eingabe und einen Programmbaum als Ausgabe erhalten. Der String repräsentiert in diesem Fall das zu parsende Programm. Die Ausgabe ist der entwickelte Programmbaum, der die Struktur der Eingabe widerspiegelt. In Haskell könnte der Datentyp wie folgt beschrieben werden:

```
type Parse = String -> Tree
```

Ein Parser besteht aus einer Vielzahl von vielen elementar Parsern die wiederum durch kombinatoren verbunden sind, wobei jede dieser Kombinationen immer nur eine Teilstruktur des gesamten Systems verarbeiten. Das Problem hierbei ist, die Zwischenspeicherung der Teilergebnisse (die Teilbäume) und die noch zu verarbeitenden Symbole. Es gibt keine Möglichkeit die Daten in einer globalen Datenstruktur zu speichern (auf die Verwendung von Monaden sei hier verzichtet). Daher müssen die ermittelten Resultate im Funktionsergebnis zwischengespeichert werden. Eine verfeinerte Version der Implementierung kann so aussehen:

```
type Parse = String -> ( Tree, String )
```

Betrachtet man den Aspekt des gegenseitigen Aufrufs der einzelnen Parser erneut, so wird der Fall eintreten, dass ein Parser kein, ein oder sogar mehrere Ergebnisse liefert. Bei der bisherigen Betrachtung ist immer nur ein Ergebnis vorgesehen. Ein String zum Beispiel könnte mehrere Interpretationswege zulassen oder es kann der Fall eintreten, dass kein Ergebnis gefunden wird. Hierfür ist es notwendig, eine Liste von Ergebnissen in `Parse` zuzulassen. Eine Möglichkeit ist, dass das Ergebnis als Liste von Ergebnissen gespeichert wird. Jedes Element der Liste enthält den Teilbaum und die noch zu verarbeitenden Symbole. Wird eine leere Liste zurückgeliefert, so wurde kein Ergebnis erzielt.

```
type Parse = String -> [ ( Tree, String ) ]
```

Wenn nur das erste Ergebnis der Liste benötigt wird, so ist lediglich der Kopf der Liste zu verwenden. Aufgrund von *Lazy evaluation* wird es keine Effizienzverluste geben. Die nicht verwendeten Ergebnisse werden erst bei Gebrauch berechnet.

Der Datentyp `Parse` wird im `Tree` den erzeugten Syntaxbaum und im `String` die noch zu verarbeitenden Symbole speichern. Die Struktur des Syntaxbaumes ist allerdings von Anwendungsfall zu Anwendungsfall unterschiedlich. Als Beispiel wird eine Baumstruktur für arithmetische Ausdrücke oder in einem anderen Fall für eine XML-Datenstruktur benötigt. Damit die Baumstruktur so flexibel und unabhängig vom Anwendungsfall ist, muss ein weiterer Abstraktionsprozess durchgeführt werden.

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

Bislang ist der Parser auf Strings beschränkt. Strings werden in Haskell als eine Liste von Characters implementiert. Es ist allerdings denkbar, dass es zu einem späteren Zeitpunkt anstatt einer Zeichenkette Tokens von einem Lexer als Eingabe gibt. Für diesen Fall ist der Typ ein letztes mal zu modifizieren.

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

oder etwas lesbarer auch so:

```
type Parse symbol result = [symbol] -> [ ( result, [symbol] ) ]
```

Dieser Datentyp wird in diesem Kapitel und in den folgenden Beispielen verwendet.

3.3. Elementar Parser

3.3.1. symbol

Nachdem der Datentyp `Parse` entwickelt ist, kann der erste kleine Parser entwickelt werden, der das Symbol `a` erwartet. Entspricht das erste Symbol nicht dem `a`, so wird eine leere Liste zurück geliefert.

```
> symbola :: Parse Char Char
> symbola [] = []
> symbola (x:xs)
> | x== a = [ ( a ,xs) ] > | otherwise = []
```

Im Hugs aufgerufen, sieht das Ergebnis wie folgt aus. Im ersten Beispiel wurde das Symbol `a` erkannt und als Ergebnis abgelegt. Im zweiten Aufruf, entspricht das erste Symbol dem Zeichen `b`, entsprechend wurde die leere Liste zurückgegeben.

```
? symbola "abc"
[( a , "bc")]
```

```
? symbola "bcd"
[]
```

Die Funktion ist wie erwartet.

Damit nicht für jedes verwendete Symbol ein eigener Parser implementiert werden muss, ist dieser zu verallgemeinern. Hierfür ist der Parser `symbol` abgebildet.

```
> symbol :: Eq a => a -> Parse a a
> symbol s [] = []
> symbol s (x:xs)
> | s==x = [ (x,xs) ]
> | otherwise = []
```

Bei dem Parser `symbol` wird das zu überprüfende Symbol als Parameter übergeben. Zusätzlich wurde eine Verallgemeinerung des Symboltyps vorgenommen. Es muss sich hierbei nicht mehr wie im vorherigen Beispiel um ein `Character` handeln. Es muss lediglich sichergestellt sein, dass für den `Typ` eine Instanz der Klasse `Equality` vorliegt.

3.3.2. spot

Eine weitere wichtige Parse Funktion ist `spot`. Der Parser `spot` erwartet als Parameter eine Funktion. Diese wird auf den Kopf der Symbolliste angewendet und liefert entsprechend eine leere Liste oder eine Liste von möglichen Ergebnissen.

```
> spot          :: (a-> Bool) -> Parse a a
> spot f []     = []
> spot f (x:xs)
>   | f x       = [(x,xs)]
>   | otherwise = []
```

Durch die Implementierung von `spot` ist es jetzt möglich eine wesentlich elegantere Version von `symbol` zu schreiben.

```
> symbol :: Eq a => a -> Parse a a
> symbol t = spot (==t)

? symbol 'a' "abc"
[('a',"bc")]
```

3.3.3. dig

Dieser Parser erwartet eine Zeichenkette und überprüft, ob es sich bei dem ersten Symbol um eine Ziffer handelt.

```
> dig  :: Parse Char Char
> input = (spot isDigit) input
```

3.3.4. succeed

Ein einfacher allerdings in der Bedeutung nicht zu verachtender Parser ist `succeed`. `succeed` liefert immer einen `Parse` mit der übergebenen Eingabe zurück. Die Parameter werden nicht verarbeitet oder in irgendeiner Form verändert.

```
> succeed :: b -> Parse a b
> succeed val inp = [(val,inp)]
```

Diese Funktion kann zum Beispiel als “*epsilon*–” Funktion verwendet werden. Die Funktion liefert immer einen leeren Programmbaum und die übergebenen Symbole zurück. Verwendet wird diese Funktion später bei den Parser kombinatoren.

3.4. Parser kombinatoren

Im vorherigen Kapitel 3.3 über elementar Parser sind nur Parser für terminal Symbole entwickelt worden. Viel interessanter sind jedoch Parser, die terminal Symbole und nichtterminal Symbole verarbeiten. Hierfür werden sogenannte Parser kombinatoren benötigt. Im folgenden werden einige Kombinatoren behandelt. Zum Beispiel ein Parser zum Sequentialisieren zweier Parser (erst P1 dann P2) oder eine Alternative (P1 oder P2).

3.4.1. Die Alternative

Die Alternative verarbeitet zwei übergebene Parser und liefert als Funktionsergebnis eine Liste von möglichen Ergebnissen.

```
> infixr 4 <|>
> (<|>) :: Parse a b -> Parse a b -> Parse a b
> (p1 <|> p2) input = (p1 input) ++ (p2 input)
```

Die Alternative ist hier als <|> Operator mit der 4. Prioritätsstufe und einer rechts Assoziativität implementiert. Grund für die Implementierung als Operator ist die bessere lesbarkeit des Quellcodes. Der Parser hätte natürlich auch als normale Funktion implementiert werden können. Betrachtet man zunächst die Funktionsimplementierung. Im Funktionsrumpf werden lediglich die beiden Parser p1 und p2 auf den input angewendet. Die Ergebnisse der beiden Parser werden durch den Listenkonkatenationsoperator ++ verbunden. Zum Beispiel:

```
? (dig <|> symbol 'a') "abc"
-- [] ++ [('a',"bc")]
[('a',"bc")]
```

In dem Beispiel schlägt der Parser `dig` fehl aber der Parser `symbol 'a'` liefert ein Ergebnis.

```
? (dig <|> symbol 'a') "123"
-- [('1',"23")] ++ []
[('1',"23")]
```

```
? (symbol 'a' <|> symbol 'b') "123"
-- [] ++ []
[]
```

3.4.2. Die Sequenzialisierung

Ein weiterer wichtiger Parser ist die Sequenzialisierung. Dieser Parser wendet zunächst den Parser `p1` auf den `input` an und der Rest wird auf `p2` angewendet. Liefert `p1` und `p2` ein Ergebnis, so ist das Sequenzergebnis ein Tupel von Ergebnissen. Das Ergebnis der Sequenz ist ein Parser. Hierdurch kann das endgültige Resultat durch einen weiteren Kombinator verarbeitet werden.

```
> infixr 6 <*>
> (<*>)      :: Parse a b -> Parse a c -> Parse a (b,c)
> (p1 <*> p2) input = [( (x1, x2), xs2 )
>                       | (x1, xs1) <- p1 input
>                       , (x2, xs2) <- p2 xs1
>                       ]
```

Die Funktions- und Arbeitsweise der Sequenz ist durch die folgenden Beispiele verdeutlicht.

```
? (symbol 'a' <*> symbol 'b') "123"
[]
```

```
? (symbol 'a' <*> symbol 'b') "abc"
[(('a', 'b'), "c")]
```

```
? (symbol 'a' <*> symbol 'b' <*> symbol 'c') "abcd"
[((('a', ('b', 'c')), "d")]
```

Bei dem ersten Beispiel ist das Ergebnis des ersten Parsers `symbol 'a'` eine leere Liste. Im zweiten Beispiel sind beide Parser erfolgreich. Näher betrachtet wird das letzte Beispiel. Das Ergebnis ist ein Baum mit Tupeln vom Typ `[((Char, (Char, Char)), [Char])]`.

```
? ( symbol 'a' <*> ( symbol 'b' <*> symbol 'c') ) "abcd"
~ [ ( ('a', ('b', 'c')), "d" ) ]
= [( (x1, x2), xs2 )
  | (x1, xs1) <- [ ('a', "bcd" ) ]
  , (x2, xs2) <- [ ('b', 'c'), "d" ) ]
  ]
```

```
? ( symbol 'b' <*> symbol 'c' ) "bcd"
~ [ ('b', 'c'), "d" ) ]
= [( (x1, x2), xs2 )
  | (x1, xs1) <- [ ('b', "cd" ) ]
  , (x2, xs2) <- [ ('c', "d" ) ]
  ]
```

3.4.3. Der Umwandler

Einer der wichtigsten Parser kombinatoren ist der Umwandler. Der Umwandler ist zum Anwenden einer Funktion `f` auf einen Parser `p`. Hiermit ist es zum Beispiel möglich, eine Ziffer als Character in ein Integer umzuwandeln.

```

> infixr 5 <@
> (<@)      :: Parse a b -> ( b -> c ) -> Parse a c
> (p <@ f) input = [ (f x,xs)
>                   | (x,xs) <- p input
>                   ]

```

Zunächst wird als Parameter ein Parser `p` und eine Funktion `f` erwartet. Der Parser `p` wird auf den `input` angewendet, die Funktion `f` wird auf das Resultat `x` angewendet. Das Ergebnis ist ein neuer Parser vom Typ `Parse a c`. Der Typ `c` ist hierbei der Resultattyp von der Funktion `f`.

```

> digit :: Parse Char Int
> digit = spot isDigit <@ f
>   where f c = ord c - ord '0'

```

```

? digit "123"
[(1,"23")]

```

Wie Eingangs erwähnt, kann durch den Operator `<@` eine Ziffer in ein Integer umgewandelt werden. Die Funktion `digit` stellt eine mögliche Verwendung des Umwandlers dar.

3.4.4. many

Zahlen zum Beispiel bestehen in den seltensten Fällen nur aus einer Ziffer. Damit diese Liste von Objekten zu einem Symbol umgewandelt werden kann, ist ein Parser `many` notwendig.

```

> many  :: Parse a b -> Parse a [b]
> many p = (p <*> many p <@ list)
>         <|>
>         (succeed [])
>   where list (x,xs) = x:xs

```

Es gibt zwei unterschiedliche Funktionsergebnisse:

- Die Liste kann leer sein (`succeed []`).
- Die Liste ist nicht leer und enthält ein Symbol gefolgt von einer Liste von Objekten (`p <*> many p`). Damit der Parser `many` die Liste verarbeitet ruft sich dieser rekursiv auf. Das Ergebnis muss noch vom Typ `(x,xs)` in den Typ `(x:xs)` umgewandelt werden. Hierfür kann der Umwandler `<@` verwendet werden. Durch den Umwandler wird die Liste von Curried zu Uncurried umgewandelt.

In den folgenden Beispielen wird deutlich, dass jede *gültige* Kombination in der Liste gespeichert wird. Der Parser `many` liefert somit mehr als ein Ergebnis. Dieses ist unbedingt bei der Verarbeitung zu beachten.

```

? many (spot isDigit) "123abc"
[("123","abc"),("12","3abc"),("1","23abc"),(,"","123abc")]

```

```

? many (spot isAlpha) "test"
[("test",""),("tes","t"),("te","st"),("t","est"),(,"","test")]

```

3.4.5. option

Ein weiterer Parser ist die `option`. Zum Beispiel kann eine Zahl negativ oder positiv sein. Die negative Zahl ist durch das voranstehende “-” gekennzeichnet.

```
> option      :: Parse a b -> Parse a [b]
> option p input = ( ( p <@ (:[ ]) ) )
>               <|>
>               ( succeed [ ] ) ) input
```

Die `option` wendet jeweils zwei Parser auf den `input` an. Zum einen den Parser `p` und den Umwandler `<@`. Zum anderen den Parser `succeed []`. Durch diese Alternative Verknüpfung werden bei Erfolg mehr als ein Ergebnis geliefert. Siehe dazu das folgende Beispiel:

```
? option (symbol '-') "-123"
[("-", "123"), ("", "-123")]
```

```
? option (symbol '-') "123"
[("", "123")]
```

3.5. Ein Parser für arithmetische Ausdrücke

3.5.1. Die Idee

Nachdem nun diverse elementar Parser und Parser kombinatoren entwickelt wurden, sollen diese nun verwendet und benutzt werden. Hierfür soll ein Taschenrechner entwickelt werden. Bei dem folgenden Beispiel handelt es sich um einen *recursive decent parser*. Der Taschenrechner soll ganzzahlige Operationen berechnen.

- Zahlen: “12” und “-88”, negative Zahlen werden durch “-” dargestellt
- Operatoren: “+”, “-”, “*”, “/”, “%”
- “%” steht für die modulo Operation
- Nur ganzzahlige Division
- Leerzeichen sind nicht erlaubt
- Jeder Ausdruck muss geklammert sein! Zum Beispiel kann ein Ausdruck wie folgt aussehen: “(23-(20/2))”. Das bedeutet es wird eine natürlich strukturierte Grammatik verwendet. Durch die Klammerung jedes Ausdrucks werden Mehrdeutigkeiten vermieden.

3.5.2. Die Grammatik

Zur Verständlichkeit sei die natürlich strukturierte Grammatik in einer BN-Form gegeben.

```
Expr ::= const | '(' Expr Op Expr ')'
Op    ::= '+' | '-' | '*' | '/' | '%'
```

In Haskell ist die Grammatik als einfacher arithmetischer Datentyp implementiert. Ein Ausdruck kann aus einer natürlichen Zahl `Lit` oder einem Ausdruck `Bin` bestehen. `Bin` ist aber wiederum ein Operator und zwei Zahlen.

```
data Expr = Lit Int | Bin Op Expr Expr
data Op   = Add | Sub | Mul | Div | Mod
```

Der Ausdruck `"(14+-2)"` wird dann wie folgt dargestellt `"Bin Add Lit 14 Lit (-2)"`. Ein etwas kompliziertere Ausdruck ist zum Beispiel `"(120*(20/2))"` und wird so dargestellt: `"Bin Mul Lit 120 (Bin Div Lit 20 Lit 2)"`.

3.5.3. Der Parser

Nachdem die Grammatik durch einen einfachen arithmetischen Datentyp in Haskell beschrieben wurde, müssen nur noch die entsprechenden Parser an die Grammatik angepasst werden.

```
> parser :: Parse Char Expr
> parser = litParse <|> opExprParse
```

Der `parser` beschreibt eine *Expression*. Die *Expression* besteht aus einem `Literal` Parser oder einer `opExpr` Parser.

```
> litParse :: Parse Char Expr
> litParse
>   = (
>     option (symbol '-') <*> many1 (spot isDigit)
>   ) <@ charListToExpr.join
>   where join = uncurry (++)
```

```
? litParse "14"
[(Lit 14,""),(Lit 1,"4")]
```

```
? litParse "-4"
[(Lit (-4),"")]
```

Der Parser `litParse` erstellt aus der übergebenen Symbolliste die möglichen *Literals*.

```
> opExprParse :: Parse Char Expr
> opExprParse = (symbol '(' <*>
>               parser <*>
>               spot isOp <*>
>               parser <*>
>               symbol ')')
>               ) <@ makeExpr
>   where
>   makeExpr :: (a, (Expr, (Char, (Expr, b)))) -> Expr
>   makeExpr (_, (e1, (bop, (e2, _)))) = Op (charToOp bop) e1 e2
```

Durch den Parser `opExprParse` werden rekursiv die Ausdrücke zusammengebaut.

Beispiele:

```
? parser "(14+-2)"  
[(Bin Add (Lit 14) (Lit (-2)), "")]
```

```
? parser "(14-2)+a"  
[(Bin Sub (Lit 14) (Lit 2), "+a")]
```

Wie das letzte Beispiel zeigt, werden noch keine Fehler abgefangen. Es müsste in einem nächsten Entwicklungsschritt überprüft werden, ob die Menge der noch zu prüfenden Symbole leer ist. Dann würde der letzte Fehler auch entdeckt werden.

4. Parsec

Für Haskell gibt es eine Vielzahl von Parsern. Hier sei nur der bottom-up Parser *Happy* und der top-down Parser *Parsec* erwähnt. Im folgenden wird eine kleine Einführung in *Parsec* vollzogen.

4.1. Was ist Parsec?

Parsec ist eine mächtige Parser Bibliothek. Es handelt sich hierbei um einen top-down Parser unter der Verwendung von Monaden. Im Allgemeinen wird *Parsec* als LL[1]-Parser betrieben. Backtracking ist auch möglich und wird im Kapitel 4.2.3 ausführlich behandelt. *Parsec* ermöglicht die Ausgabe von aussagekräftigen Fehlermeldungen bei syntaktischen Fehlern. Desweiteren lässt sich mit *Parsec* eine lexikalische Analyse und das Parsen in einem realisieren. Es ist nicht wie häufig notwendig, jeweils zwei unterschiedliche Tools zu verwenden (z.B. *Lex* und *Yacc*).

Bei weiterführendem Interesse ist der interessierte Leser auf das Referenzenhandbuch von *Parsec* verwiesen. Ein Verweis auf die Internetseite findet sich im Anhang.

4.2. Beispiele mit Parsec

Bevor *Parsec* in einem Modul verwendet werden kann, muss die Bibliothek im Haskellmodul bekannt gemacht werden.

```
> module Main where
> import Parsec
```

Nachdem *Parsec* importiert ist, sind die ersten einfachen Beispiele möglich. Die Funktion `many` ist aus dem Kapitel 3.4.4 schon bekannt. In *Parsec* ist diese Funktion ebenfalls implementiert. Der folgende Parser `simple1` erwartet eine Liste von Buchstaben. Die verwendete Funktion `letter` ist in *hugs* vordefiniert.

```
> -- Ein Parser fuer Identifier
> simple1 :: Parser [Char]
> simple1 = many letter
```

Die Funktionsweise sei anhand des folgenden Aufruf verdeutlicht. Die Funktion `parseTest` ist von *Parsec* und dient zum einfachen Testen von Parsern.

```
? parseTest simple1 "abc123"
"abc"
```

4.2.1. Die Sequenz und Auswahl

Wie schon im Kapitel 3.4 über Parser kombinatoren, wurde deutlich wie wichtig ein Operator für Sequenzen und einer für die Auswahl ist. Durch die Verwendung von Monaden in Parsec, muss für die Sequenz kein extra Operator erstellt werden. Es wird der Monadenoperator `do` für Sequenzen verwendet.

Der folgende Parser `openClose0` erwartet ein Klammernpaar.

```
> openClose0 :: Parser Char
> openClose0 = do{ char '('
>                ; char ')'
>                }
```

Allerdings ist ein Klammernpaar nicht sonderlich spannend. Eine weitaus spannendere Version eines Klammernpaarparsers könnte wie folgt aussehen.

```
openClose1 :: Parser ()
openClose1 = do{ char '('
                ; openClose1
                ; char ')'
                ; openClose1
                }
<|> return ()
```

Durch den eigenen rekursiven Aufruf erwartet `openClose1` eine Sequenz von Klammernpaaren. Der `<|>` Operator ist die Auswahl. Wird keine öffnende Klammer gefunden, so wird `return()` aufgerufen und die Rekursion beendet.

4.2.2. Parser mit Semantik

Ein ähnlicher Parser wie `openClose1`, bloss das dieser die Anzahl der geschweiften Klammernpaare berechnet. Das Ergebnis der rekursiven Aufrufe wird in den Variablen `m` und `n` gespeichert (Achtung: single Assignment). Zum Abbruch der Rekursion wird die Alternative `<|>` verwendet. Wird keine öffnende Klammer `"{"` gefunden, so wird `return 0` ausgeführt.

```
> noOfBrace = do{ char '{'
>                ; m <- noOfBrace
>                ; char '}'
>                ; n <- noOfBrace
>                ; return (1+m+n);
>                }
> <|> return 0
```

```
? parseTest noOfBrace "{}{}"
2
```

```
? parseTest noOfBrace "{{}}{}"
3
```

4.2.3. Die Auswahl

Wie in der Einleitung von Kapitel 4.1 erwähnt, handelt es sich bei *Parsec* um einen LL[1]-Parser. Was aber bedeutet LL[1]? Bei der Topdown-Analyse wird ausgehend von der Wurzel eine Reihe von Ableitungsschritten gesucht und ausgeführt. Ohne die Verwendung von Backtracking muss jede Ableitungsfolge in einer Grammatik eindeutig ausgewählt werden können.

Im folgenden soll die Problematik des Backtracking behandelt werden. Es ist zunächst der folgende Ausschnitt einer Grammatik bekannt. Terminalsymbole sind hierbei großgeschrieben. Klein geschrieben sind die nichtterminal Symbole.

```
cond ::= IF boolexpr THEN stmt FI
      | IF boolexpr THEN stmt ELSE stmt FI
```

Zusätzlich ist die folgende Symbolfolge (Anweisung) zu verarbeiten:

```
IF (a>b) THEN RETURN a ELSE RETURN b FI
```

Es wird im folgenden betrachtet, wie die gegebene Anweisung nach dem Topdown-Verfahren verarbeitet werden würde. Ein Symbolzeiger zeigt auf das nächst zu bearbeitende Symbol IF. Ein Parserzeiger “befindet” sich gerade vor der Produktionsregel `cond`. Bei der Verarbeitung der Produktionsregeln wird zunächst fälschlicherweise die erste Alternative “`IF boolexpr THEN stmt FI`” ausgewählt. Hierbei werden die Terminalsymbole verglichen und schrittweise Syntaxbäume für die nichtterminal Symbole `boolexpr` und `stmt` aufgebaut. Allerdings schlägt ein abschließender Vergleichstest mit `FI` aus der Grammatik und dem `ELSE` aus der Symbolfolge fehl. Es muss nun der erzeugte Teilbaum verworfen werden und der Zeiger auf die Symbolfolge wieder an den Anfang gesetzt werden. Der Parserzeiger wird auf die zweite Alternative von `cond` gesetzt werden. Jetzt kann die zweite und richtige Ableitungsfolge verarbeitet werden. Es werden entsprechend die Terminalsymbole verglichen und für `boolexpr` und `stmt` Teilbäume erstellt.

Bei dieser Vorgehensweise und der gegebenen Grammatik ist das Problem, dass unter Betrachtung des ersten Symbols `IF` keine Aussage getroffen werden kann, welche der beiden Ableitungsregeln angewendet werden soll.

Die gleiche Problematik soll in *Parsec* mit einer überschaubaren Grammatik behandelt werden. Eine `condition` besteht aus drei Symbolen der öffnenden Klammer, dem `a` oder `b` und einer schließenden Klammer. Bei einer gegebenen Symbolfolge von `(b)` kann keine Aussage getroffen werden, ob die erste oder zweite Regel ausgewertet werden soll.

```
cond ::= (a) | (b)
```

Eine mögliche Implementierung mit *Parsec* könnte wie folgt aussehen:

```
> or0 = string "(a)"
>     <|>
>     string "(b)"
```

`string "(a)"` ist hierbei ein eigenständiger Parser der eine Symbolfolge “(a)” erwartet. Die Alternative `<|>` versucht nur den zweiten Parser `string "(b)"`, wenn der erste Parser keine Symbole verarbeitet hat.

```
? parseTest or0 "(a)"
"(a)"
```

```
? parseTest or0 "(b)"
parse error at (line 1, column 1):
unexpected "b"
expecting "(a)"
```

Das erste Beispiel wird wie erwartet ausgeführt. Das zweite Beispiel bricht mit einer Fehlermeldung ab. Der Parser `string "(a)"` überprüft und vergleicht Symbol für Symbol und erwartet anstelle des `b` ein `a`. Der zweite alternative Parser `string "(b)"` wird nicht überprüft, weil der erste schon Symbole verarbeitet hat. Die zweite Alternative wird somit nie ausgeführt. Eine verbesserte Version von `or0` sei wie folgt abgebildet:

```
> or1 = do { char '('
>           ; char 'a' <|> char 'b'
>           ; char ')'
>           ; return "ok"
>           }
```

```
? parseTest or1 "(a)"
"ok"
```

```
? parseTest or1 "(b)"
"ok"
```

Funktion- und Arbeitsweise von `or1` ist wie erwartet. Der erste Parser überprüft das erste Symbol, die öffnende Klammer. Der zweite Parser ist entweder ein `a` oder `b` gefolgt von der schließenden Klammer. Für dieses indirekte “vorrasschauen” (`a`) oder (`b`) gibt es in *Parsec* einen extra Parser `try`. `try` hat die gleiche Funktion wie die oben verwendete Technik im Parser `or1`. Zum verdeutlichen ist das gleiche Beispiel unter Verwendung von `try` abgebildet.

```
> or2 = try( string "(a)" )
>       <|>
>       try( string "(b)" )
```

Es sei an dieser Stelle darauf hingewiesen, dass in einem Parser unter der Verwendung durch `try` das Backtracking implementiert werden kann. Es muss aber deutlich sein, dass dieses Zeit und Performance kostet.

4.2.4. Ein Taschenrechner mit Parsec

Der Vollständigkeit halber wird auch in diesem Kapitel ein kleiner Taschenrechner mit *Parsec* dargestellt. Es wird allerdings auf die Entwicklung und Funktionsbeschreibung verzichtet.

```
module MyPCalc1 where

import Parsec
import ParsecExpr

expr :: Parser Integer
```

```

expr = buildExpressionParser table factor
      <?> "expression"

table :: OperatorTable Char () Integer
table = [
  [binary "*" (*) AssocLeft, binary "/" div AssocLeft],
  [binary "+" (+) AssocLeft, binary "-" (-) AssocLeft]
]
  where binary s f assoc
        = Infix( do{ string s; return f} ) assoc

factor = do { char '('
            ; x <- expr
            ; char ')'
            ; return x
            }
      <|> number
      <?> "simple expression"

number :: Parser Integer
number = do {
  ds <- many1 digit
  ; return (read ds)
  }
      <?> "number"

```

Beispielaufufe:

```
? parseTest expr "2+5*2" -- '*' hat hoehere Prioritaet
12
```

```
? parseTest expr "(6+2)*3"
12
```

```
? parseTest expr "48/2/2" -- '/' ist links Assoziativ
12
```

5. Download

Diese Dokumentation steht in den folgenden Formaten zur Verfügung:

- Postscript (doku.ps) Beste Qualität zum Drucken.
- Acrobat (doku.pdf) Gute Qualität zum Online Anschauen, mit verlinktem Inhaltsverzeichnis

Alle in dieser Dokumantation verwendeten Beispiele finden sich zum Runterladen als:

- tar-Archiv (exampleParser.tar.gz)

Die verwendeten Folien für meinen Vortrag gibts hier:

- Acrobat (PDF) Für Beamer und *Acrobatreader* im Vollbildmodus.

Literaturverzeichnis

- [Bird] R. Bird:
Introduction to Functional Programming using Haskell,
Prentice Hall, ISBN 0-13-484346-0, 1998
- [Erwig] G. Erwig:
Übersetzerbau,
Springer Verlag, ISBN 3-540-65389-9, 1999
- [Fokker] J. Fokker:
Functional Parsers. Lecture Notes of the Baastad Spring school on Functional Programming,
<http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps>, Mai 1995
- [Leijen] D. Leijen:
Parsec, a fast combinator parser,
<http://www.cs.uu.nl/~daan/parsec.html>, 4 Oktober 2001
- [Thompson] S. Thompson:
Haskell - The Craft of Functional Programming,
Springer Verlag, ISBN 0-201-34275-8, 1999
- [Wirth] N. Wirth:
Compilerbau,
Teubner Studienbücher, ISBN 3-519-32338-9, 1986