Parsen für Linguisten und andere Programmierlaien

Sven Eric Panitz

10. Januar 1996

1 Willkommen

[1]

Sie wissen, was eine kontextfreie Grammatik ist? Sie wissen, was eine Funktion ist? Sie wissen, welche Menge durch den Ausdruck $\{(x, y)|x \in A, y \in B\}$ beschrieben wird? Sie können nicht programmieren? Sie wissen nicht, was ein Algorithmus ist? Sie glauben, Sie werden nie Computerprogramme verstehen? Dann sind Sie hier richtig. Wir laden Sie ein: Lernen Sie, ein Computerprogramm zum Parsen eines Strings nach einer Grammatik zu schreiben.

2 Wir parsen: Auf wieviel Arten kann der Satz von der Grammatik erzeugt werden?

Einen String gemäß einer kontextfreien Grammatik zu parsen, ist eine der fundamentalsten Aufgaben in der Informatik. Fast jedes Programm, das Daten aus einer Textdatei einliest, benötigt einen Parser, der diesen Text in einen internen Datentypen umwandelt. Die Aufgabe, einen Parser zu schreiben, ist in imperativen Sprachen hinreichend erforscht. Verschiedenste Algorithmen zum Parsen einer Sprache, z.B. Zerteilertabellen oder rekursiv absteigende, sollten aus den Grundvorlesungen bekannt sein (z.B. [Deu87]). Besonders bequeme Gemüter bedienen sich zum Parsen eines Werkzeuges wie YACC. Ein tatsächlich implementierter Parser ist allerdings für einen Programmierlaien nicht mehr zu verstehen, auch wenn dieser generell einen Algorithmus zum Parsen kennt.

Um die hohe Abstraktionsebene einer nicht-strikten funktionalen Sprache (hier Gofer) zu demonstrieren, soll hier ein Programm zur Konstruktion eines Parsers aufgeführt werden, das auch z.B. ein Linguist ohne große Kenntnisse

[1] Ich danke Norbert Klose für seine Durchsicht.

von Programmiersprachen verstehen kann. Seien wir gespannt, ob dieses gelingen wird.

Wir haben als Ziel einen rekursiv absteigenden Parser im Auge, der ausgehend von einer kontextfreien Grammatik alle möglichen Ableitungen ausprobiert, in wieweit sie zur Ableitung des Eingabestrings führen können. Dieser Parser wird für jede Produktion der Grammatik genau eine Funktion bereitstellen. Ausgabe dieses Parsers soll für uns zunächst einmal entweder eine Art `fail`-Ausgabe, wenn der String mit der vorgegebenen Grammatik nicht ableitbar ist, oder aber eine Liste aller möglichen Ableitungen sein. Da der `fail`-Fall gerade der Fall ist, in dem es keine mögliche Ableitung gibt, können wir diesen Fall mit der Ausgabe einer leeren Liste abfangen. Dieses entspricht auch der schon seit langem praktizierten Identifizierung der leeren Liste mit dem boolschen `false` in Lisp. Wir folgen nun zunächst den in [Wad85] vorgestellten Methoden.

Schreiben wir zum Warmwerden eine erste kleine Funktion: Stellen wir uns eine Liste von mit Symbolen assoziierten Elementen (diese könnten ja ein bestimmtes Lexikon darstellen) vor. Wir wollen eine Funktion haben, die zu einer solchen Liste und einem String die Liste aller mit diesem String assoziierten Elemente ausgibt (z.B. alle Lexikoneinträge für ein bestimmtes Stichwort). Eine solche Funktion ist in Gofer extrem einfach zu definieren:

```
>assoc xys x = [y |  (x',y) <- xys, x'==x ]
```

Das Hinschreiben dieser Funktionsdefinition ist wohl kaum noch als Programmieren zu bezeichnen. Für den Laien wollen wir die obige Zeile trotzdem in eine natürlichsprachliche Formulierung übersetzen:

> Die Anwendung der Funktion `assoc` auf zwei Argumente `xys` und `x` ist die Liste aller Symbole `y`, so daß es in der Eingabeliste `xys` ein Paar der Art `(x,y)` gibt, dessen erstes Element `x` und dessen zweites das Symbol `y` ist.

Wir können uns im Goferinterpretierer auch den Typ der Funktion `assoc` angeben lassen:

```
Eq a => [(a,b)] -> a -> [b]
```

Natürlichsprachlich würden wir diese Typinformation wie folgt umschreiben:

> Angenommen wir haben einen Typ `a`, auf dem eine Gleichheitsfunktion definiert ist, dann ist die Funktion `assoc` eine Funktion, die für die Eingabe einer Liste von Paaren zweier Elemente, das erste vom Typ `a`, das zweite von einem beliebigen Typ `b` und der Eingabe eines zweiten Elements vom Typ `a` eine Ergebnisliste von Elementen des Typs `b` berechnet.

Diese Funktion werden wir gegen Ende des Aufsatzes noch häufig benutzen.

Wir wollen uns jetzt dem Parsen einer kontextfreien Grammatik nähern. Wir fangen ganz klein an, mit einer bestimmten Grammatik, die sehr primitiv ist und nur eine bestimmte reguläre Sprache erzeugen kann:

$$
\begin{array}{lcl}
Z & \rightarrow & \epsilon \\
Z & \rightarrow & AZ \\
A & \rightarrow & \alpha \\
 & & \alpha \in \{a, b, c\}
\end{array}
$$

Dieses Grammatik erzeugt die Sprache aller Buchstabenketten aus den Buchstaben a,b und c. Sie besteht dabei streng genommen aus 5 Produktionen. Wir können jetzt für jede dieser Produktionen versuchen, eine Funktion zu definieren, die testet, ob ein eingegebenes Wort mit dieser Produktion abgeleitet werden kann. Dieses ist für die Produktionen, die ein Terminalsymbol einführen, sehr einfach:

```
>lit x "" = []
>lit x (x':xs)
>   |x==x'     = [xs]
>   |otherwise = []
```

Wir wollen auch diese Funktionsdefinition für Leute übersetzen, die die stringente Syntax einer Programmiersprache nicht gewohnt sind:

> Die Funktion `lit` ist eine Funktion mit zwei Eingabeargumenten. Sie unterscheidet zunächst, ob das zweite Argument die leere Zeichenkette ist (`""`), oder einen ersten Buchstaben und eine restliche Zeichenkette hat (`(x':xs)`). Im Falle der leeren Zeichenkette gibt sie eine leere Liste aus. Im Falle einer Zeichenkette mit einem Buchstaben als erstes Element, wird unterschieden, ob dieser Buchstabe gleich der Eingabe für das erste Element ist oder nicht. Ist dieses nicht der Fall, wird wieder eine leere Liste ausgegeben, sonst wird als Funktionsergebnis eine Liste mit einem Element konstruiert. Dieses Element ist der Restteil des zu parsenden Strings ohne das erkannte Terminalwort.

Am leichtesten läßt sich die Funktion wohl verstehen, wenn wir ein paar Beispielergebnisse für sie angeben:

```
lit 'a' "apple"
["pple"]
```

```
lit 'b' "apple"
[]

lit 'a' ""
[]
```

Wir haben hier, wie oben erwähnt, die Fälle, ob eine Regelanwendung Erfolg hat oder nicht, mit der Ausgabe einer leeren oder nichtleeren Liste unterschieden.

Wir haben noch ein weiteres Beispiel für eine Regel, die terminierend ist und kein weiteres Nichtterminal auf der rechten Seite stehen hat. Auch für diese Regel ist eine einfache Funktion zu definieren:

```
>epsilon xs = [xs]
```

Für diese Funktion gibt es keinen *fail*-Fall. Sie liefert immer ein Ergebnis. Dieses ist die Liste aus dem zu parsenden String, da ja nur das leere Wort erkannt werden konnte. Wir haben jetzt für die Terminalproduktionen Funktionen geschrieben. Die Produktionen einer kontextfreien Grammatik sind zumeist etwas komplizierter. Zunächst kann es für ein Nichtterminal mehrere Regelalternativen geben. So gibt es im obigen Beispiel 3 verschiedene Alternativen für das Nichtterminal A. Wenn wir jetzt ebenso eine Funktion schreiben wollen, die den Produktionen für das Nichterminal A entspricht, so brauchen wir ein Konstrukt, das uns die einzelnen Alternativen für A zusammenfassen läßt. Auch in der Notation von Grammatiken ist es üblich, alle Produktionen für ein Nichtterminal in einer Produktion zu schreiben, indem man die Alternativen durch ein | trennt. Die Produktion für A wäre dann im obigen Beispiel:

$$\mathrm{A} \rightarrow a|b|c$$

Wir wollen also eine Funktion schreiben, die den vertikalen Strich der Grammatik entspricht. Dieses ist aufgrund unserer Entscheidung, Listen als Ergebnis zu benutzen, extrem einfach. Wir definieren:

```
>alt p q = \xs -> (p xs) ++ (q xs)
```

Diese Funktionsdefinition besteht zwar nur aus einer Zeile, verlangt jedoch schon etwas Abstraktionsvermögen. Wir versuchen wieder, eine natürlichsprachliche Umschreibung zu geben:

> Die Funktion `alt` hat zwei Eingabeargumente. Die beiden Argumente sind jeweils Funktionen, die Produktionen in der Grammatik entsprechen. Das Ergebnis ist wieder eine Funktion.[2] Sie hat als Eingabe

[2] `\xs -> ...` ist die Schreibweise für $\lambda xs.(\ldots)$

einen String, der zu parsen ist, und gibt als Ergebnis die Liste aller Ergebnisse, die man durch das Parsen des Eingabestrings mit den Funktionen der Argumente erhält.[3]

Jetzt haben wir also schon das Werkzeug, um die Produktion für das Nichtterminal A zu formulieren.

```
>a = alt (lit 'a') (alt (lit 'b') (lit 'c'))
```

Die Wirkungsweise dieser Produktion sei auch mit einem Beispiel demonstriert:

```
a "banane"
["anane"]

a "zitrone"
[]
```

Beim Formulieren einer Produktion der Grammatik haben wir noch eine weitere Möglichkeit: es können auf der rechten Seite Wörter von Symbolen stehen. So wie in der Produktion:

$$\mathrm{Z} \rightarrow \mathrm{AZ}$$

Die rechte Seite ist hier als Sequenz von zu parsenden Teilen zu verstehen. Es soll, um ein aus dem Nichtterminal Z entstandenes Wort zu parsen, zunächst ein Wort aus dem Nichtterminal A und von dem Rest ein Wort aus dem Nichtterminal Z geparst werden. Wir müssen nun noch eine Funktion definieren, die es erlaubt, eine derartige Sequenz von Parsefunktionen zu definieren.[4] Wir können den gesuchten Sequenzoperator recht knapp definieren:

```
>seq p q = \xs -> [ xs2 | xs1<-p xs, xs2<- q xs1]
```

Diese Funktion ist zwar auch in nur einer Zeile niedergeschrieben, doch für den Laien unserer Zielgruppe etwas kompliziert. Wollen wir hoffen, daß die folgende Umschreibung mit seinem Wissen über Grammatiken und Mengenschreibweisen ausreicht, ein Verständnis für sie zu gewinnen.

[3]++ ist eine Operation, die die links und rechts von ihr stehenden Listen aneinanderhängt - die Listenkonkatenation.

[4]in der Grammatik machen wir dieses durch das direkte Nacheinanderschreiben der beiden Nichtterminale auf der rechten Seite deutlich. In unserer Programmiersprache wird das direkte Hintereinanderschreiben zweier Symbole als Funktionsanwendung benutzt; wir sind also gezwungen, uns für diese Operation ein Funktionssymbol zu definieren.

> Die Funktion `seq` hat zwei Eingabeargumente. Die beiden Argumente sind jeweils Funktionen, die Produktionen in der Grammatik entsprechen. Das Ergebnis ist wieder eine Funktion. Sie hat als Eingabe einen zu parsenden String. Die Ausgabe ist der Rest des Eingabestrings, der nach dem Parsen, zuerst nach der ersten Produktion und dann nach der zweiten, übrigbleibt.

Damit schreiben wir für die Produktionen des Nichtterminals Z die Funktion:

```
>z = alt epsilon (seq a z)
```

Wir haben nun schon alles, was wir brauchen, um einen Parser für eine beliebige kontextfreie Grammatik zu schreiben. Betrachten wir zunächst ein paar Beispiele, um die Arbeitsweise unseres bisherigen Parsers zu sehen:

```
z "aabcabc"
["aabcabc", "abcabc", "bcabc", "cabc", "abc", "bc", "c", []]

z "aabda"
["aabda", "abda", "bda", "da"]
```

Der Aufruf `z "aabcabc"` soll jetzt bedeuten: Versuche das Wort `aabcabc` aus dem Nichtterminalsymbol Z abzuleiten. Unser Programm macht aber noch etwas anderes: Es versucht, auch alle Anfänge des Eingabestrings abzuleiten. Dabei bleibt dann immer ein Reststring, der nicht geparst werden konnte, übrig. Sind wir nur daran interessiert, ob ein String abgeleitet werden konnte oder nicht, dann müssen wir alle Elemente mit nichtleerem Rest aus der Ergebnisliste ausschließen. Wir definieren eine Funktion, die uns die Anzahl der vollständigen Abarbeitungen der Eingabe aus Z gibt:

```
>zvollst = \xs -> length [x| x<-z xs, x==""]
```

Wir geben noch einmal eine natürlichsprachliche Beschreibung dieser Funktion:

> `zvollst` ist eine Funktion, die zunächst seinen Eingabestring mit der Funktion `z` parst, aus der dann entstehenden Ergebnisliste alle Elemente löscht, die noch einen vom Parser nicht abgearbeiteten Reststring darstellen, und die Länge der so entstandenen Liste als Ergebnis ausgibt.[5]

[5] `length` ist eine im Gofer-Prelude definierte Standardfunktion auf Listen

Diese Funktion können wir jetzt als unseren Parser benutzen. Betrachten wir zwei Beispielrechnungen:

```
zvollst "aaabbababababccc"
1
(548 reductions, 824 cells)

zvollst "ababbbccabccacccdaac"
0
(640 reductions, 965 cells)
```

Im Prinzip können wir jetzt mit den Funktionen `seq`, `alt`, `lit` und `epsilon` für jede kontextfreie Grammatik sofort einen Parser programmieren. Wir müssen hierzu lediglich jede Produktion der Grammatik in eine Funktion umsetzen. Dieses geschieht nach folgendem Schema:

$$\mathrm{Z} \to alt_1 | \cdots | alt_n$$
$$\Rightarrow$$
$$\texttt{z = \textbackslash xs -> (alt } alt_1 \texttt{ (alt } \cdots \texttt{(alt } alt_{n-1}\ alt_n\texttt{))} \cdots \texttt{))}$$

Hierin müssen die einzelnen Alternativen übersetzt werden. Dies geschieht mit der Sequenzoperation nach folgenden Schema:

$$\mathrm{Z}_1 \cdots \mathrm{Z}_n$$
$$\Rightarrow$$
$$\texttt{(seq z1 (seq z2} \cdots \texttt{(seq zn-1 zn))} \cdots \texttt{))}$$

[6] Wenn schließlich ein Terminalsymbol eingeführt wird, so wird hierfür die Funktion `lit` mit dem entsprechenden Symbol als Argument eingeführt. Ich empfehle an dieser Stelle, für eine Beispielgrammatik den Parser hinzuschreiben. Es werden sich dann einige der nachfolgenen Punkte von selbst motivieren.

3 Linksrekursionen oder Mondbasis $\alpha 1$ antwortet nicht

Vielleicht ist ja jemand der Aufforderung im vorherigen Abschnitt nachgekommen und hat einen Parser für eine kontextfreie Grammatik aufgestellt. Vielleicht sogar für eine Grammtik der Art:

$$\mathrm{T} \ \to \ \epsilon \mid \mathrm{T}a$$

[6] Die `zi` sind hier die Funktionen, die das Nichtterminal Z_i realisieren

und hat die folgende Funktionen erhalten:

```
>t = alt (seq t (lit 'a')) epsilon
```

Dann wird er enttäuscht festgestellt haben, daß sein Parser leider nicht wie versprochen funktioniert, sondern sich nach einem Aufruf `t "ttd"` nicht mehr meldet. Dieses liegt an der Form der Grammatik: sie ist nicht frei von Linksrekursionen. Das führt dazu, daß unser Parser, beim Versuch eine Ableitung aus dem Zeichen T zu erzeugen, immer wieder die Funktion `t` aufruft, ohne ein Stück der zu parsenden Eingabe abzuarbeiten. Er befindet sich in einer Endlosschleife. Allgemein gilt:
Sei G eine Grammatik mit Produktionen der Form:

$$X \ \rightarrow \beta \mid X\ \alpha,\ X \in N \text{ und } \alpha, \beta \in (N \cup T)^{*7} \text{ mit } \beta \neq X\beta',$$

so kann man aus X ein Wort $\beta\alpha^*$ ableiten. Dieses muß auch in einer linksrekursionsfreien Version G' noch möglich sein. Man ersetze deshalb die obigen Regeln durch:

$$\begin{aligned} X &\rightarrow \beta \mid \beta B_n \\ B_n &\rightarrow \alpha \mid \alpha B_n \end{aligned}$$

mit B_n als neuem Nichtterminalzeichen. Derartige Ersetzungen werden so lange durchgeführt, bis keine Linksrekursionen mehr vorhanden sind.
In unserem obigen Beispiel erhalten wir dann die Grammatik:

$$\begin{aligned} T &\rightarrow \epsilon \mid \epsilon T_1 \\ T_1 &\rightarrow a \mid aT_1 \end{aligned}$$

Wir erhalten dann einen neuen Parser, der wie gewünscht funktioniert:

```
>t' = alt epsilon (seq epsilon t1)

>t1 = alt (lit 'a') (seq (lit 'a') t1)
```

Wir haben hier ein schönes Beispiel dafür, wie sehr wir uns beim Programmieren losgelöst haben von programmiersprachlichen Details und uns auf problemspezifische Dinge konzentrieren können. Wir haben nicht den Parser geändert sondern die Spezifikation.

[7] N bezeichnet die Menge aller Nichtterminale und T die Menge aller Terminale

4 Nicht nur destruktiv - auch konstruktiv

Die Entscheidung, ob ein String zu einer – durch eine Grammatik spezifizierten – Sprache gehört oder nicht, ist für ein Problem, das ein Programm lösen soll, nur der erste Teil der Aufgabe. Für die meisten Anwendungen ist es nötig, das durch den String dargestellte Wort in eine interene Syntaxdarstellung zu übersetzen, den Syntaxbaum, der dann z.B. für einen Kompilierer oder einen Beweiser zur weiteren Verarbeitung benötigt wird. Ein Linguist hat vielleicht eher im Sinn, nach Montegue jeden syntaktischen Ableitungsschritt mit einer semantischen Bedeutungsfunktion zu koppeln. Bisher haben wir die geparsten Teile unseres Eingabestrings einfach nur fortgeworfen. Jetzt wollen wir versuchen, aus ihnen eine sinnvolle Ausgabe des Parsers zu konstruieren. Wir wollen hierzu sowohl die Konstruktion von Syntaxbäumen, als auch die Konstruktion einer Montegue Grammatik betrachten.

4.1 Syntaxbäume

Wir wollen den Eingabestring parsen und als einen Syntaxbaum darstellen. Hierbei wird eine Satzstruktur in Form eines Baumes erstellt, der ein übergeordnetes Symbol an einem Knoten und die diesem direkt untergeordneten Symbole an den Söhnen des Knotens stehen hat. Eine häufige Interpretation dieses Baumes wird sein, daß ein Knoten ein Funktionssymbol darstellt und seine Söhne die Argumente dieser Funktion. Wir brauchen einen Zieltypen, auf den der geparste String abgebildet werden soll. Im Folgenden sei exemplarisch ein Typ für arithmetische Ausdrücke mit drei auf ihnen definierten Funktionen gegeben:

```
>data Arith = Zahl
>           | Summe Arith Arith
>           | Produkt Arith Arith
>           | Faku Arith
```

Wir wollen unseren fachfremden Laien nicht aus den Augen verlieren und versuchen, ihm auch diese Programmzeilen zu erklären:

> Wir definieren einen Datentypen, von dem wir fortan Daten als Ergebnis von Funktionen konstruieren oder Daten als Argument an Funktionen übergeben können. Dieser Datentyp sei unter dem Namen `Arith` bekannt. Ein Datum des Datentyps `Arith` kann vier verschiedene Ausprägungen haben:
>
> - es kann aus dem einfachen Symbol `Zahl` bestehen,
> - es kann aus dem Symbol `Summe` gefolgt von zwei weiteren Daten des Datentyps `Arith` bestehen,

- es kann aus dem Symbol `Produkt` gefolgt von zwei weiteren Daten des Datentyps `Arith` bestehen,
- es kann aus dem Symbol `Faku` gefolgt von einem weiteren Datum des Datentyps `Arith` bestehen.

Jetzt, wo wir wissen, in welcher Form die Ausdrücke unserer Sprache strukturiert werden sollen, ist es an der Zeit, diese Sprache auch zu definieren:[8]

$$\begin{array}{lcl} \mathrm{S} & \rightarrow & \mathrm{S}+\mathrm{S} \\ & | & \mathrm{S}*\mathrm{S} \\ & | & \mathrm{S} \\ & | & n \end{array}$$

Wir erinnern uns, daß wir mit dieser Grammatik garantiert Schiffbruch erleiden werden und eleminieren ganz fix die Linksrekursionen:

$$\begin{array}{lcl} \mathrm{S} & \rightarrow & f\mathrm{S} \\ & | & f\mathrm{S}\mathrm{S}_1 \\ & | & n \\ & | & n\mathrm{S}_1 \\ \mathrm{S}_1 & \rightarrow & +\mathrm{S} \\ & | & *\mathrm{S} \\ & | & +\mathrm{S}\mathrm{S}_1 \\ & | & *\mathrm{S}\mathrm{S}_1 \end{array}$$

Für diese Grammatik ist nun also der Parser zu konstruieren. Das Ergebnis dieses Parsers soll jetzt eine Liste von Paaren sein. Das erste Element dieser Paare soll das erzeugte Datum des Datentyps `Arith`, das zweite soll der nicht geparste Reststring unserer Eingabe sein. Wir müssen zu dem Parser, den wir bisher schon bauen können, noch Teile zur Konstruktion der ersten Komponente des Ausgabepaares erzeugen. Diese erste Komponente soll vom Typ `Arith` sein.
Beginnen wir mit den Funktionen, die Terminale erkennen. Zu jedem Terminal soll jetzt ein Paar konstruiert werden. Beginnen wir mit dem Terminal n:

```
>n "" = []
>n ('n':xs) = [(Zahl,xs)]
>n _ = []
```

[8] die Terminalzeichen der Sprache sind $\{+,*,f,n\}$

Die übrigen drei Terminale konstruieren auf folgende Weise ihr Ergebnis:

```
>f "" = []
>f ('f':xs) = [(\x-> Faku x,xs)]
>f _ = []

>su "" = []
>su ('+':xs) = [(\x y -> Summe x y,xs)]
>su _ = []

>p "" = []
>p ('*':xs) = [(\x y -> Produkt x y,xs)]
>p _  = []
```

Wir sehen, daß diese drei Funktionen alle die gleiche Struktur haben. Es läßt sich natürlich auch wieder eine Funktion zur Konstruktion der Funktionen für die Terminale aufstellen ähnlich zu `lit`, allerdings mit dem zusätzlichen Argument, welches Ergebnis in der ersten Komponente des Paares erzeugt werden soll:

```
>lit' _ _ [] = []
>lit' erg c (x:xs)
>    |c == x     = [(erg,xs)]
>    |otherwise = []
```

Wir umschreiben diese Funktion:

> Die Funktion `lit'` hat drei Argumente. Das bei erfolgreichen Parsen zu erstellende Ergebnis, das Terminal, das erkannt werden soll und der String, der geparst werden soll. Ist der erste Buchstabe des Strings gleich dem Terminal, dann wird mit dem Ergebnisargument und dem Reststring ein Ausgabepaar konstruiert, andernfalls wird als *fail* die leere Liste ausgegeben.

Damit lassen sich die Funktionen für Terminale natürlich einfacher schreiben. Z.B.:

```
>su' = \xs -> lit' (\x y -> Summe x y) '+' xs
```

Wir können auch für alle unsere Terminale Tabellen aufstellen, in denen ihre Semantik beschrieben ist. In diesen Tabellen wird jedes Terminalzeichen mit einer Semantik assoziiert.

```
>tab0 = [('n',Zahl)]

>tab1 = [('f',\x -> Faku x)]

>tab2 = [('+',\x y -> Summe x y), ('*',\x y -> Produkt x y)]
```

Wir können jetzt die Tabellen für die Semantik zur Definition unserer Terminalsymbole benutzen. Dabei erinnern wir uns unserer allerersten Funktion, die uns half, Tabelleneinträge abzufragen:

```
>su'' "" = []
>su'' (x:xs)
>  |x == '+'  = map (\y -> (y,xs)) (assoc tab2 '+')
>  |otherwise = []
```

Wir können für die Semantiktabellen natürlich auch eine Version unserer `lit`-Funktion angeben und daraus die einzelnen Funktionen der Terminale instanzieren:

```
>lit'' _ _ [] = []
>lit'' tab c (x:xs)
>   |c == x    = map (\y -> (y,xs) )(assoc tab c)
>   |otherwise = []

>su''' = \xs -> lit'' tab2 '+' xs
```

Wir brauchen drei verschieden Tabellen, da wir Terminale haben, die semantisch drei verschiedene Typen repräsentieren:

- Konstanten vom Typ `Arith`
- einstellige Funktion vom Typ: `Arith -> Arith`
- zweistellige Funktion vom Typ: `Arith -> Arith-> Arith`

Es liegt daher nahe, für diese drei verschiedenen Typen von Terminalen auch eigene Nichtterminalklassen bereitzustellen. Wir erhalten dann die modifizierte Grammatik:

$$S \rightarrow \mathrm{Typ}_1 S$$

$$
\begin{array}{rcl}
 & | & \mathrm{Typ}_1\mathrm{S}\mathrm{S}_1 \\
 & | & \mathrm{Typ}_0 \\
 & | & \mathrm{Typ}_0\mathrm{S}_1 \\
\mathrm{S}_1 & \to & \mathrm{Typ}_2\mathrm{S} \\
 & | & \mathrm{Typ}_2\mathrm{S}\mathrm{S}_1 \\
\mathrm{Typ}_0 & \to & n \\
\mathrm{Typ}_1 & \to & f \\
\mathrm{Typ}_2 & \to & + \mid *
\end{array}
$$

Jetzt können wir unsere kleine Arithmetiksprache sehr einfach um weitere Funktionssymbole erweitern. Wir haben jetzt Nichtterminalsymbole, die nur noch durch Terminale entsprechend der Tabelle instanziiert werden müssen. Wir können hierfür einfach eine allgemeine Funktion, die das nächste Zeichen in der entsprechenden Tabelle sucht, definieren:

```
>lookup tab = \xs -> lookaux xs
>                 where
>                   lookaux [] = []
>                   lookaux (x:xs) = lit'' tab x (x:xs)

>typ0 = lookup tab0
>typ1 = lookup tab1
>typ2 = lookup tab2
```

Doch halten wir uns nicht weiter mit den Terminalen auf und betrachten die beiden Funktionen, die wir bei der Konstruktion komplexerer Produktionen benötigen. Hier gibt es zunächst eine erfreuliche Erkenntnis: Die Funktion zur Konstruktion der Alternativen in einer Produktion können wir unverändert weiterbenutzen. Wir schreiben sie zu Erinnerung nochmals auf:

```
>alt' p q = \xs -> (p xs) ++ (q xs)
```

Anders sieht es hingegen mit der Sequenzfunktion aus. Wir hatten für die Sequenz benötigt, daß wir das Ergebnis einer Funktion für eine Produktion als Eingabe für eine weitere solche Funktion nehmen können. Dieses geht jetzt nicht mehr, da wir als Ergebnis nicht mehr nur Reststrings haben, sondern Paare, von denen uns nur die zweite Komponente als Eingabe dienen kann. Dieses ist bei der Sequenzbildung natürlich zu berücksichtigen:

```
>seq' p q = \xs ->
>  [ ((erg1,erg2),xs2) | (erg1,xs1) <- p xs, (erg2,xs2) <- q xs1 ]
```

Wir sammeln also zunächst die Einzelergebnisse der beiden – in der Sequenz beteiligten – Teilableitungen in einem Paar auf. Wann und in welcher Weise wir diese Einzelergebnisse zu einem Gesammtergebnis zusammenführen können, ist noch nicht spezifiziert. Die Frage nach dem *Wann*, läßt sich einleuchtend mit: „Immer wenn eine Produktionsalternative abgearbeitet wurde“ beantworten. Die Frage des *Wie*, läßt sich allgemein nicht beantworten. Es soll ja damit die Semantik des geparsten Ausdruck festgelegt werden, die Teil der Spezifikation ist. Wir können aber ein Werkzeug definieren, das es uns erleichtert, die Semantikinformation in die Produktionsalternativen hineinzucodieren. Bisher ist das Ergebnis einer Funktionsalternative eine Liste von Paaren. Die erste Komponente dieser Paare stellt die aufgesammelten Ergebnisse unseres Parsedurchlaufs dar. Auf diese muß jetzt noch eine Funktion angewendet werden, die die Teilergebnisse zusammenfügt. Wir definieren uns eine Funktion, die uns diese Anwendung auf die erste Komponenten durchführt.

```
>sem p f = \xs -> [(f x,y) | (x,y) <- p xs]
```

Hier brauchen wir vielleicht etwas Verständnishilfe:

> `sem` ist eine Funktion, die zwei Eingabeargumente hat. Das erste `p` ist eine Funktion für eine Produktion der Grammatik, das zweite `f` ist eine Funktion, die beschreibt, wie die in dieser Produktion erzeugten Teilergebnissse kombiniert werden sollen. Das Ergebnis von `sem` ist wieder eine Funktion. Sie parst einen String, indem sie zunächst die Funktion `p` auf den zu parsenden String anwendet und dann für alle enstandenen Ergebnisse die Funktion `f` auf die erste Komponente anwendet.

Jetzt gilt es, am Beispiel die semantischen Regeln in die Grammatik einzubauen. Wir haben als Gerüst für den Parser:

```
s = alt'
       (seq' typ1 s)
       (alt'
           (seq' typ1 (seq' s s1))
           (alt' typ0 (seq' typ0 s1)))

s1 = alt' (seq' typ2 s) (seq' typ2 (seq' s s1))
```

In diesen Programmrumpf[9] flicken wir jetzt die Semantikinformation ein:

```
>s = alt'
>       (sem (seq' typ1 s)             (\(a,b) -> a b))
>            (alt'
>                 (sem (seq' typ1 (seq' s s1))
>                                      (\(a,(b,c))-> (c (a b))))
>                 (alt' typ0
>                      (sem (seq' typ0 s1)
>                                      (\(a,b) ->  b a) )))

>s1 = alt'
>        (sem (seq' typ2 s)            (\(a,b) -> a b))
>        (sem (seq' typ2 (seq' s s1))
>                                      (\(a,(b,c))-> a (c b)))
```

Ich gebe zu: in diesem Abschnitt wurde doch allerhand Gehirnschmalz gefordert, aber kein schreckliches Herumgefriemel mit programmiersprachlichen Eigenarten. Wenn wir dazu bedenken, daß wir nicht nur bereits ein recht abstraktes Problem gelöst haben, sondern sogar eine Anleitung zum Lösen allgemeiner solcher Probleme gegeben haben, dann dürften auch die letzten Skeptiker von der hohen Abstraktionsebene nicht-strikter funktionaler Programmiersprachen überzeugt sein. Ein so entstehender Parser hat noch eine ganze Reihe von schwerwiegenden Nachteilen. In den nachfolgenden Abschnitten werden wir uns um diese kümmern.

4.2 lazy user oder das dolce vita

Schauen wir uns die wenigen Zeilen unserer kontextfreien Grammatik an und vergleichen sie mit dem erstellten Parser, so müssen wir uns leider eingestehen, daß wir mehr tippen müssen und das Programm nicht mehr so übersichtlich notiert ist, wie die Grammatik es war. Das liegt im wesentlichen daran, daß wir Sequenzen und Alternativen aus mehr als zwei Teilen geschachtelt schreiben müssen und sie nicht so elegant implizit geklammert und in Infixschreibweise wie in der Grammatik darstellen können. Dem können wir aber abhelfen, denn Haskell erlaubt uns, als „syntaktischen Zucker“ die Definition von Infixoperatoren:

```
>infixr 8 ...
>infixl 6 ==>
>infixr 4 |||
```

[9] Achtung dieses ist keine richtige Funktionsdefinition, sondern nur die Struktur unseres Parsers

```
>(|||) = alt'
>(==>) = sem
>(...) = seq'
```

Umschrieben bedeuten diese Zeilen:

> Wir definieren drei Infixoperatoren `|||`, `==>` und `...`. Sie haben verschiedene Prioritäten in ihrer Bindung und sind implizit rechts bzw. links geklammert. Sie sollen entsprechend definiert werden als die Funktionen `alt'`, `sem` bzw. `seq'` in Infixschreibweise. Statt `alt' a (alt' b c)` können wir jetzt schreiben: `a ||| b ||| c`.

Damit sieht unser obiger Parser dann gleich schon viel handlicher aus:

```
>s'  = typ1 ... s'          ==>  (\(a,b) -> a b)           |||
>      typ1 ... s' ... s1'  ==>  (\(a,(b,c))-> c (a b))    |||
>      typ0                                                 |||
>      typ0 ... s1'         ==>  (\(a,b) -> b a)

>s1' = typ2 ... s'          ==>  (\(a,b) -> a b)           |||
>      typ2 ... s' ... s1'  ==>  (\(a,(b,c)) -> a (c b))
```

Dieses ähnelt nun schon sehr der kontextfreien Grammatik. Man bedenke dabei, daß hier ja auch schon die semantische Information des geparsten Wortes mit spezifiziert ist, was in der Grammatik nicht der Fall ist.

4.3 schneller, höher, weiter

Wir können an unserem Parser auch in puncto Effizienz noch etwas verbessern. Diese Verbesserung wird wieder nicht die Art, in der der Parser aus der Grammatik erzeugt wird, betreffen, sondern eine Transformation der Grammatik. In beiden Produktionen haben Regelalternativen auf der rechten Seite ein gleiches Anfangswort. Versucht unser Parser vom Startsymbol alle Ableitungen zu erzeugen, so wird er an diesen Regeln zweimal dieselben Ableitungsteile erzeugen. Wir können diese doppelte Arbeit vermeiden, indem wir dieses Anfangswort für beide Alternativen zunächst zum Parsen benutzen, und dann erst parallel verschiedene Ableitungen bearbeiten. Dieses ist zu erreichen, durch Linksfaktorisieren der Grammatik. Wir erhalten als neue Grammatik:

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{F}_2|\mathrm{F}_3 \\
\mathrm{S}_1 &\rightarrow \epsilon
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & |\mathrm{F}_1 \\
\mathrm{F}_1 & \rightarrow & \mathrm{Typ}_2\mathrm{SS}_1 \\
\mathrm{F}_2 & \rightarrow & \mathrm{Typ}_1\mathrm{SS}_1 \\
\mathrm{F}_3 & \rightarrow & \mathrm{Typ}_0\mathrm{S}_1 \\
\mathrm{Typ}_0 & \rightarrow & n \\
\mathrm{Typ}_1 & \rightarrow & f \\
\mathrm{Typ}_2 & \rightarrow & +|*
\end{array}
$$

Dieses können wir wieder in einen Parser übertragen:

```
>ss  = f2       |||
       f3

>ss1 = epsilon' |||
       f1

>f1 = typ2 ... ss ... ss1  ==>  (\(a,(b,c)) -> a (c b))

>f2 = typ1 ... ss ... ss1  ==>  (\(a,(b,c)) -> a (c b))

>f3 = typ0 ... ss1         ==>  (\(a,b) -> b a)

>epsilon' xs = [(id,xs)]
```

Wir mußten uns noch eine neue Funktion für Epsilon-Produktionen definieren, da wir in dieser für die Konstruktion der Ausgabe noch eine neutrale Komponente vorsehen müssen. Da `ss1` als Ausgabe eine Funktion des Typs `Arith -> Arith` konstruiert, nehmen wir als erste Tupelkomponente einfach die Identitätsfunktion.

Wer nicht glaubt, daß wir hiermit eine Kostensenkung in der Laufzeit erreicht haben, der probiere doch einfach ein paar Beispielaufrufe aus:

```
? length [x|(x,[])<-(ss "n+n+n+n*n*n+n*n*n+n*n")]
16796
(2689538 reductions, 6670796 cells, 103 garbage collections)

?  length [x|(x,[])<-(s "n+n+n+n*n*n+n*n*n+n*n")]
16796
(27703102 reductions, 50897473 cells, 589 garbage collections)
```

Im obigen Beispielaufruf sehen wir, daß das Wort auf 16796 verschiedene Arten geparst werden kann. Das erzeugte Ergebnis dieser verschiedenen Ableitungen kann auch unterschiedlich sein. Unsere Grammatik ist hochgradig mehrdeutig. Wie solche Mehrdeutigkeiten durch implizite Klammerung aufgelöst werden kann, sei hier nicht weiter behandelt.

4.4 Montegue - wieder was für unseren linguistischen Leser

Wir scheinen, mit unserem arithmetischen Beispiel die Linguisten vergessen zu haben. Dem ist nicht ganz so. Es mögen sich Linguisten erinnern, wie Montegue die natürlichsprachliche Semantik definiert hat: er hat für jeden syntaktischen Ableitungsschritt einen semantischen Schritt konstruiert. Genau das haben wir oben auch gemacht, als wir mit jedem Ableitungsschritt einen Konstruktionsschritt auf unseren zu erstellenden Syntaxbaum gekoppelt haben. Der Schritt zu einer Semantik für eine natürlichsprachliche Syntax sollte demnach nicht viel schwerer sein. Wir werden uns dabei an die Methoden aus [FL89] halten. Versuchen wir es nun für die folgende Grammatik:

$$\begin{array}{rcl} Satz & \rightarrow & nominalPhrase\ verbalPhrase \\ verbalPhrase & \rightarrow & Verb\ nomPhrase \\ nomPhrase & \rightarrow & Substantiv \\ & & |Artikel\ Substantiv \\ & & |nomPhrase\ Konjunktion\ nomPhrase \\ nominalPhrase & \rightarrow & Name \end{array}$$

Wir haben aus Platzgründen die Terminalzeichen aus der Grammatik fortgelassen und stellen hierfür direkt Tabellen auf:

```
>tabname =[("Giuliani",[1]),("Paganini",[2]),("Carulli",[3]),
>          ("Beethoven",[4]),("Bach",[5])]

>tabsubstantiv = [("Gitarrist",[1,2,3]),("Komponist",[1..5]),
>                 ("Geiger",[2,5]),("Pianist",[4]),
>                 ("Cembalist",[5])]

>tabverb:: [(String, [Int]-> [Int] -> Bool)]
>tabverb = [("ist",teilmenge),("war",teilmenge)]

>tabkonjunktion = [("und",intersection),("oder",union)]
```

```
>tabartikel = [("ein",id)]
```

Die Terminalzeichen sind hier nicht mehr einzelne Buchstaben, sondern ganze Strings. Das wird aber die Konstruktion des Parsers in keiner Weise verkomplizieren. Wir brauchen jetzt noch die üblichen Mengen- und Listenoperationen:

```
>union:: [Int]->[Int]->[Int]
>union as bs = as ++ (bs \\ as)

>intersection:: [Int]->[Int]->[Int]
>intersection as bs = as \\ (as \\ bs)

>teilmenge:: Eq [a] => [a] -> [a] -> Bool
>teilmenge as bs = (as \\ bs) == []

>flatten :: [[a]] -> [a]
>flatten [] = []
>flatten (x:xs) = x ++ (flatten xs)
```

Damit ist die Semantik der Terminalsymbole beschrieben. Wir müssen jetzt nur noch den Parser gemäß der Grammatik niederschreiben [10], und Semantikregeln mit Ableitungsregeln koppeln:

```
>satz :: [String] -> [(Bool,[String])]
>satz = np ... vp       ==>  (\(a,b) -> b a)

>vp :: [String] -> [([Int] -> Bool,[String])]
>vp = verb ... n       ==>  (\(a,b) -> (\x -> a x b))

>n  = substantiv                                    |||
>     artikel ... substantiv   ==>  (\(a,b) -> b)   |||
>     substantiv ... n1        ==>  (\(a,b) -> b a) |||
>     artikel ... substantiv ... n1
>                              ==>  (\(a,(b,c)) -> c b)

>n1 = konjunktion ... n  ==>  (\(a,b)-> a b)  |||
>     konjunktion ... n ... n1
>                       ==> (\(a,(b,c))->(\x -> c (a b x)))
```

[10] dabei werden wir natürlich auch die Linksrekursion eleminieren

```
>np = name

>name        = lookup tabname
>substantiv  = lookup tabsubstantiv
>verb        = lookup tabverb
>artikel     = lookup tabartikel
>konjunktion = lookup tabkonjunktion
```

Wir definieren uns zum Austesten des Systems eine Anfragefunktion:

```
>tt x = [a|(a,[])<-satz (words x)]
```

Wir werden sehen, daß unsere Grammatik wieder nicht eindeutig ist und ein Satz unter Umständen verschiedene Interpretationen hat:

```
tt "Bach ist ein Komponist oder ein Gitarrist und Pianist"
[True, False]
(6803 reductions, 9323 cells)
```

Bevor wir unsere Leser mit dem Auftrag entlassen, eigene natürlichsprachliche Systeme mit der hier vorgestellten Methode zu entwickeln, wollen wir das System noch so erweitern, daß es nicht nur angibt, ob ein Satz richtig oder falsch ist, sondern auch Fragen beantworten kann. Hierzu müssen wir die Grammatik für Fragen spezifizieren:

$$Frage \rightarrow Fragepronomen\ Substantiv\ verbalPharse$$

Wir wollen es bei dieser Frageart im Beispiel belassen. Dabei können wir ganz schnell das entsprechende Programm schreiben:

```
>frage = fragepronomen ... substantiv ... vp
>                                 ==> (\(a,(b,c)) -> a b c)

>tabfragepron =
>  [("Welcher",(\x f->flatten (map (wahr f) (map (\a->[a]) x))))]
>                      where wahr f x |f x = x |otherwise = []

>fragepronomen = lookup tabfragepron
```

```
>ff fr =
> map (map head)(map ints2namen [x|(x,[])<-frage (words fr)])

>ints2namen y = map (assoc (map (\(a,[b])-> (b,a)) tabname))  y
```

Wir können auch hier eine mehrdeutige Anfrage stellen:

```
ff "Welcher Komponist war ein Komponist oder ein Gitarrist und
Pianist"
[["Giuliani", "Paganini", "Carulli", "Beethoven", "Bach"],
["Beethoven"]]
(6975 reductions, 9906 cells)
```

Damit wollen wir es belassen, bevor es nun doch unübersichtlich wird. Ich hoffe, die Programmierlaien sind noch dabei und der Leser findet etwas Spaß daran, selber ein paar natürlichsprachliche Systeme zu programmieren.

Literatur

[Deu87] P. Deussen. Einführung in die Informaik III. Skript zur Vorlesung Informatik III, Universität Karlsruhe, Fakultät für Informatik, 1987.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional languages. *The Computer Journal*, 32(2):108–121, 1989.

[Wad85] Phil Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.