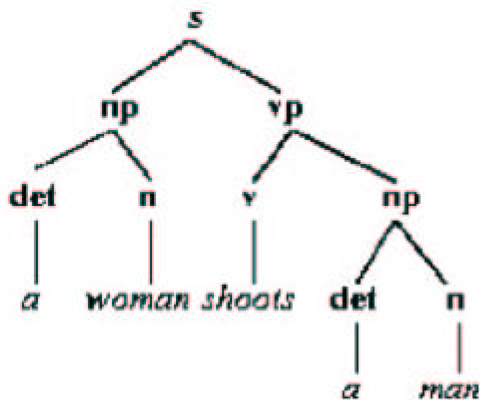
Einführung in PROLOG 7

Kontextfreie Grammatiken

Der Erschaffer von PROLOG, Alain Colmerauer, war ein Computer-Linguist. Daher ist die Computer-Linguistik immer noch das klassische Anwendungsfeld von PROLOG.
Betrachten wir zunächst kontextfreie Grammatiken, das heißt, kurz gesagt, eine endliche Sammlung von Regeln, die uns sagt, wann ein Satz zu einer Grammatik gehört (also syntaktisch korrekt ist) und wie die grammatikalische Struktur überhaupt aussieht.
Ein kurzes Beispiel für einen Teil der englischen Sprache sähe folgendermaßen aus:

s → np vp
np → det n
vp → v np
vp → v
det → ***a***
det → ***the***
n → ***woman***
n → ***man***
v → ***shoots***

Nicht-terminale Symbole sind hier: **s** für Satz, **np** für Nomenphrase, **vp** für Verbphrase, **det** für Artikel. Dazu gehören terminale Symbole (Das Alphabet) in ***kursiv***.
Dazu kommen insgesamt 9 Regeln, die uns sagen, wie Sätze der Sprache zusammengebaut werden.
Der Satz *a woman shoots a man*.ist grammatikalisch korrekt. Seine Struktur erkennen wir an folgendem **Parse**-(oder **Ableitungs-**)**Baum**:



Kontextfreie Grammatiken erkennen mit append

PROLOG ist so entworfen, dass wir die Grammatik einfach übernehmen können.
Wir wollen Sätze durch Listen repräsentieren. Der string: *a woman shoots a man* wird also dargestellt durch [a,woman,shoots,a,man].
Die Regel s -> können wir lesen als:
Eine Liste von Wörtern ist eine s-Liste, wenn sie das Resultat der Verknüpfung einer np-Liste mit einer vp-Liste ist.
Damit sieht unsere Grammatik also folgendermaßen aus:

```
s(Z) :- np(X), vp(Y), append(X,Y,Z).
np(Z) :- det(X), n(Y), append(X,Y,Z).
vp(Z) :- v(X), np(Y), append(X,Y,Z).
vp(Z) :- v(Z).

det([the]).
det([a]).

n([woman]).
n([man]).
v([shoots]).
```

Jetzt können wir das Programm verwenden, um Sätze zu erkennen:

```
?- s([a,woman,shoots,a,man]).
yes
```

aber auch, um Sätze zu erzeugen. Unsere kleine Grammatik kann 20 Sätze erzeugen, hier sind die ersten 5:

```
?- s(X).
X = [the,woman,shoots,the,woman] ;
X = [the,woman,shoots,the,man] ;
X = [the,woman,shoots,a,woman] ;
X = [the,woman,shoots,a,man] ;
X = [the,woman,shoots]
```

Wir müssen auch nicht nur Fragen über Sätze stellen, sondern können auch andere grammatikalische Kategorien betrachten:

```
?-np([a,woman]).
yes
```

Es ergibt sich allerdings das Problem, dass das Programm für die Suche nicht auf den Inputsatz zurückgreift. Wenn wir den Tracer verwenden für die Query: *s([a,man,shoots])* sehen wir, dass das Programm Nomen- und Verbphrase "rät", dann überprüft, ob man diese zum gewünschten Satz kombinieren kann.
Das Problem ist offensichtlich, dass die goals np(X). und vp(Y) mit uninstantiierten Variablen als Argumenten aufgerufen werden..
Was passiert, wenn wir die Regeln ändern, so dass append das erste goal wird?

```
s(Z) :-append(X,Y,Z), np(X), vp(Y).
np(Z) :-append(X,Y,Z), det(X), n(Y).
vp(Z) :-append(X,Y,Z), v(X), np(Y).
vp(Z) :-v(Z).

det([the]).
det([a]).

n([woman]).
n([man]).
v([shoots]).
```

Nun verwenden wir append im die Inputliste aufzusplitten, wodurch X und Y instantiiert werden.
Allerdings verwendet das Programm sehr häufig append, vor allem append mit zwei uninstantiierten Variablen als erste beide Argumente. Im vorletzten Kapitel haben wir die Ineffizienz dieses Vorgehens bereits kennengelernt. Betrachten wir nur die Analyse des Satzes *a woman shoots a man* im Tracer so stellen wir fest, dass die meisten Aufruf nicht dem Ziel dienen, den Satz zu erkennen, sondern um Listen zu bearbeiten. Wenn wir eine „normal" große Grammatik verwenden würden, hätten wir ein Problem.
Wir müssen uns also einen noch besseren Ansatz suchen.

Kontextfreie Grammatiken erkennen mit verschiedenen Listen

Eine effizientere Implementation ist das Verwenden verschiedener Listen, eine sehr beliebte Technik in PROLOG.
Die Kernidee ist, dass wir die Informationen über die grammatikalischen Kategorien nicht in einer Liste darstellen, sondern als Unterschied von zwei Listen. Anstatt also *a woman shoots a man* als [a,woman,shoots,a,man] darzustellen, verwenden wir das Listenpaar:

[a,woman,shoots,a,man] [].

Betrachten wir die erste Liste als das, *was wir verwenden wollen* (also die Input-Liste) und die zweite Liste, als das was wir verwerfen wollen (also die output-Liste). Damit repräsentiert *[a,woman,shoots,a,man] [].* den Satz *a woman shoots a man,* denn es steht für: *Wenn ich alle Symbole links verwende und alle rechts verwerfe, habe ich den Satz,der mich interessiert.*

Das heißt:Der Satz, der uns interessiert ist die Differenz des Inhalts der beiden Listen.
Diese Repräsentation ist nicht eindeutig. *a woman shoots a man* kann natürlich auch durch *[a,woman, shoots, a, man, yodeldi ,yodeldü] [yodeldi, yodeldü]* repräsentiert sein.

Dadurch erhalten wir folgendes Programm zum Erkennen unser Grammatik:

```
s(X,Z) :-np(X,Y), vp(Y,Z).
np(X,Z) :-det(X,Y), n(Y,Z).
vp(X,Z) :-v(X,Y), np(Y,Z).
vp(X,Z) :-v(X,Z).
det([the|W],W).
det([a|W],W).
n([woman|W],W).
n([man|W],W).
v([shoots|W],W).
```

Die erste Regel lautet: *Ich weiß, dass das Listenpaar (X,Y) einen Satz repräsentiert, wenn ich (1)X unter Verwerfen eines Y verwenden kann, und dann das Paar (X,Y) eine Nomenphrase repräsentiert, und ich (2) dann weiterhin Y unter Verwerfen von Z verwenden kann und das Paar (Y,Z) eine Verbphrase repräsentiert.*

Das wirkt auf den ersten Blick ein wenig komplizierter, als unsere vorherigen Versionen, aber wir haben einen immensen Vorteil gewonnen: Wir haben nicht append verwendet!
Wie verwenden wir jetzt unser Programm, um einen Satz zu erkennen?

```
?- s([a,woman,shoots,a,man],[]).
yes
```

Oder um alle Sätze der Grammatik zu erhalten

```
s(X,[]).
```

Wenn wir uns die erste Query im Tracer ansehen, erkennen wir die Effizienz unseres Vorgehens.
Es wäre natürlich perfekt, ein Programm zum Erkennen von Grammatiken zu haben, das so effizient, wie dieses ist, aber so einfach zu verstehen, wie die ersten beiden.
Unter Verwenden von **Definiten Klauselgrammatiken** gelingt uns dieses auch.

Definite Klauselgrammatiken – Definite Clause Grammar (DCG)

Ein erstes Beispiel

Unsere Grammatik sähe als DCG folgendermaßen aus:
s --> np,vp.
np --> det,n.
vp --> v,np.
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
Das entspricht unser ersten Beschreibung und kann exakt so in PROLOG geladen werden. Aber wie verwenden wir DCG? Exakt genauso wie wir unser Programm mit den verschiedenen Listen verwendet haben. Wollen wir z.B. herausfinden, ob *a woman shoots a man* ein Satz ist, starten wir die Query:
?- s([a,woman,shoots,a,man],[]).
Wollen wir alle Sätze der Grammatik ausgegeben bekommen:
?- s(X,[]).
Wenn wir herausfinden wollen, ob *a woman* eine Nomenphrase ist:
?- np([a,woman],[]).
usw.

Woran liegt das? Einfach ausgedrückt: Diese DCG **ist** unser Listen-Erkenner. PROLOG übersetzt intern diese User-freundliche Notation in ein ähnliches Programm wie das letzte oben.

Rekursive Regeln

Unsere kleine Grammatik kann 20 Sätze erzeugen. Es ist jedoch nicht schwer eine Grammatik zu schreiben, die unendlich viele Sätze erzeugen kann: Wir benötigen einfach nur rekursive Regeln. Fügen wir folgende Regeln der Grammatik hinzu:

s → s conj s
conj → ***and***
conj → ***or***
conj → ***but***

Damit können wir so viele Sätze, wie wir mögen zusammenfügen zu Sätzen wie z.B. *The woman shoots the man or the man shoots the woman but the man shoots the man*

Also müssen wir nur die Regeln:

s --> s,conj,s.
conj --> [and].
conj --> [or].
conj --> [but].

zu unser DCG hinzufügen. Aber wie arbeitet PROLOG dann mit einer solchen DCG?
Fügen wir zuerst einmal die neuen Regeln am Anfang ein, vor der Regel
s -> np,vp. Wenn wir jetzt die Query *s([a,woman,shoots],[])* starten gerät PROLOG in eine Endlosschleife.
Wenn PROLOG versucht die neue erste Regel zu erfüllen, trifft es immer wieder auf die rekursive Definition.
Wenn wir die rekursive Regel **s -> s,conj,s** am Ende der Grammatik einfügen, scheint alles gut zu funktionieren. Was passiert aber, wenn wir die Query *s([woman,shoot],[]),* starten, also einen Satz, der nicht von unser Grammatik akzeptiert wird?
Hier gerät PROLOG wieder in eine Endlosschleife.
Anders als in unseren bisherigen rekursiven Definitionen bestimmt in unserem jetzigen Programm die Reihenfolge der goals auch die Reihenfolge der Worte im Satz. Daher können wir nicht einfach so diese Reihenfolge ändern Fall, denn ob unsere Grammatik den Satz *the woman shoots the man and the man shoots the woman* (s -> s,conj,s) oder den Satz *and the woman shoots the man the man shoots the woman* (s -> conj,s,s) akzeptiert, macht einen Unterschied.
Die einzige Lösung ist, ein neues nichtterminales Symbol einzuführen, z.B. simple_s für einen Satz ohne angefügte weitere Sätze. Unsere Grammatik sieht dann folgendermaßen aus:

s --> simple_s.
s --> simple_s conj s.
simple_s --> np,vp.
np --> det,n.
vp --> v,np.
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
conj --> [and].
conj --> [or].
conj --> [but].

DCG sind leider nicht allmächtig. Sie helfen bei der Notation, aber man kann leider nicht immer davon ausgehen, man müsse die Grammatik einfach als DCG aufzuschreiben und es funktioniert. Denn intern sind DCG normales PROLOG, so dass man sich immer wieder überlegen muss, was der PROLOG Interpreter mit ihnen anstellt.

DCG für formale Sprachen

Betrachten wir die einfache formale Sprache $a^n b^n$, die beispielsweise die Zeichenketten *ab*, *aabb*, *aaabbb* and *aaaabbbb* usw. akzeptiert (auch den leeren String).
Die kontextfreie Grammatik dafür sieht nun folgendermaßen aus:
s → ε
s → l s r
l → a
r → b
und unsere DCG:
s --> [].
s --> l,s,r.
l --> [a].
r --> [b].

Terme anzeigen

Zum Abschluss dieser Übung wollen wir noch einige eingebaute PROLG Prädikate kennenlernen, mit denen wir Terme anzeigen können.
Es gibt z.B das Prädikat **display/1** welche Terme auf dem Bildschirm anzeigt:

?-display(bald(ist,weihnachten)).
Anzeige unten: bald(ist,weihnachten)

Genaugenommen zeigt PROLOG die interne Repräsentation des Terms an:
?-display(2+3+4).
Anzeige unten: **+(+(2, 3), 4)**

Damit kann man sehr gut lernen, wie Operatoren in PROLOG funktionieren, aber schöner wäre es natürlich die benutzerfreundliche Scheibweise anzuzeigen, besonders bei Listen.
Das erledigt das eingebaute Prädikat w**rite/1**

?-write(2+3+4).
Anzeige unten 2+3+4

?-write(.(a,.(b,[]))).
Anzeige unten [a, b]

?-write(X).
Anzeige unten: X
X = X

?-X=a, write(X).
Anzeige unten: a
X=a

?-write(a),write(b).
Anzeige unten: **ab**

?-write(a),write(" "),write(b).
Anzeige unten: **a b**

?-write(a),nl,write(b).
Anzeige unten: **a**
b

Übung

7.1

Schreiben Sie eine DCG, die Sätze in der Aussagenlogik erkennen und konstruieren kann. Zur Erinnerung: Eine kontextfreie Grammatik für die Aussagenlogik sähe z.B. folgendermaßen aus:

aus ® p
aus ® q
aus ® r
aus ® Ø aus
aus ® (aus Ù aus)
aus ® (aus Ú aus)
aus ® (aus Þ aus)

Konzentrieren Sie sich zuerst auf die DCG, d.h anstelle von *Ø(p Þ q)* akzeptieren Sie Sätze wie *[nicht, ,(' , p, darausfolgt, q, ,)']*

7.2

Ergänzen Sie ihr Programm um die Operatorendefinitionen für *und*, *oder*, *nicht* und *darausfolgt,* so dass PROLG Ausdrücke in Aussagenlogik akzeptiert:

?-display(not(p implies q)).
not(implies(p,q)).

?-display(not p implies q).
implies(not(p),q)

Bzw. welche Symbole sie immer benutzen wollen (/\, \/, => usw.)