Syntax natürlicher Sprachen
WS 2015/16
Mini-Kurs Prolog und Grammatiken

Hans Leiß

Universität München
Centrum für Informations- und Sprachverarbeitung

18. November 2015

Prolog = *Pro*gramming in *Log*ic

Wir verwenden Prolog, weil es

- eine Notation für (verallgem.) kontextfreie Grammatiken hat,
- die Grammatiken werden in lauffähige Programme kompiliert,
- Grammatikregeln werden als logische Aussagen über Wortfolgen interpretiert und vom Programm überprüft.

Welches Prolog?

- SWI-Prolog 6.6.x, eingebaut in Suse-Linux und Ubuntu-Linux,
- Installieren (Linux) mit: `sudo apt-get install swi-prolog` `http://packages.debian.org/swi-prolog`
- Das neueste SWI-Prolog 7.2 (siehe `www.swi-prolog.org/`) hat einen Datentyp der Zeichenreihen und könnte Anpassungen unserer Grammatiken/Lexika erfordern.

Ausführliche Prolog-Kurse:

- P.Blackburn e.a. `http://www.learnprolognow.org`
- Meine Hompage, WiSe 2012/13, Symbolisches Programmieren (Kursfolien, Übungsaufgaben, Musterlösungen)

Wir machen hier nur soviel von Prolog, daß Sie verstehen

- was genau ist in den Grammatikregeln erlaubt?
- in welches Prolog-Programm wird eine Grammatik übersetzt?
- wie sieht ein Prolog-Programm im allgemeinen aus?
- wie sucht Prolog eine Lösung auf eine Benutzeranfrage?

Prolog übersetzt eine Grammatik in ein Erkennungsprogramm, das nur feststellt, ob ein Eingabesatz grammatikalisch korrekt ist. Wir brauchen ein Analyseprogramm (Parser), das ggf. sagt, warum, d.h. syntaktische Analysen (Syntaxbäume) ausgibt; das gebe ich vor.

Prolog-Programme

Ein Prologprogramm P entspricht einer Reihe von Beweisregeln

$$\frac{A_1, \ldots, A_n}{B},$$

d.h. *wenn-dann*-Regeln, die besagen:

wenn A_1 bis A_n bewiesen sind, dann ist auch B bewiesen.

Bei $n = 0$ heißt B ein Axiom oder (als bewiesen geltendes) Faktum.

Ein Programm P ist wie eine logische Theorie; nachdem es geladen ist, kann der Benuzter Anfragen Q stellen, und der Prolog-Compiler versucht, Q mit Hilfe von P zu beweisen; beim Beweis werden i.a. Objektvariable x in Q durch Terme t belegt, was die Antwort bildet.

Beispiel

Programm:

$$\frac{}{a < b} \qquad \frac{}{b < d} \qquad \frac{X < Y,\ Y < Z}{X < Z}.$$

Frage: $a < U$?

Antworten: 1. $U = b$ (Vergleich mit dem Axiom $a < b$);

2. $U = d$ (Vergleich mit der Regel und Beweisen der Prämissen)

$$1.\frac{a < Y, \quad Y < U}{a < U} \qquad 2.\frac{\overline{a < b}, \quad b < U}{a < U} \qquad 3.\frac{\overline{a < b}, \quad \overline{b < d}}{a < d}$$

Frage: $V < a$

Antwort: ?

Prolog versucht es mit der Regel $\dfrac{V < Y, Y < a}{V < a}$ und muß dazu Fragen der gleichen Form ($Y < a$) beantworten, was divergiert.

In Prolog-Programmen verwendet man eine lineare Schreibweise:

⟨*Form einer Beweisregel in Prolog*⟩≡

```
B :- A1, ..., An.    % B, wenn A1 und ... und An
```

⟨*Form eines Faktums in Prolog*⟩≡

```
B :- true.           % B ist bewiesen (Axiom).

B.                   % B ist Axiom (Kurzform).
```

Wie die Aussagen $B, A_1, \ldots, A_n$ aussehen dürfen, wird noch gesagt.

Wichtig ist, daß Axiome und Regeln des Programms *angeordnet* sind und in dieser Reihenfolge beim Beweisen ausprobiert werden.

Datentypen in Prolog

Prolog hat, grob gesagt, nur ein paar Datentypen.

- Ganzzahlen (engl. Integer), $0, 1, 2, \ldots, -1, -2, \ldots$
- Atome, die Zeichenreihen aus `[a-z][0-9a-zA-Z_]*` oder `'[A-Z][0-9a-zA-Z_]*'` (mit gewissen Sonderzeichen)
- Variable, die Zeichenreihen aus `[A-Z][0-9a-zA-Z_]*` oder `_[0-9a-zA-Z_]*`
- Terme, definiert durch: (i) Jede Variable, jedes Atom und jede Zahl ist ein Term, und (ii) Sind $t_1, \ldots, t_n$ Terme und ist f ein Atom, so ist $f(t_1, \ldots, t_n)$ ein Term.

  Manche 2-stelligen Funktionssymbole f kann man infix schreiben, z.B. $s + t$ statt $+(s, t)$, und s :- t statt :-(s, t).

Der Programmierer kann eigene „Datentypen“ einführen, indem er mit einem logischen Prädikat festlegt, welche Terme als Objekte des Datentyps gelten (s.u.).

Klauseln und Programme

Ein Prädikat p/n ist ein Atom p mit einer Stelligkeit $n \geq 0$, zu dem es im Programm eine Klausel oder (Beweis-) Regel der Form

$$p(t_1, \ldots, t_n) \ \texttt{:-} \ \mathit{Rumpf}.$$

gibt, wobei $t_1, \ldots, t_n$ Terme sind und der Rumpf

- ein Term $q(s_1, \ldots, s_k)$ ist, wo q/k ein Prädikat ist, oder
- aus solchen Termen R und S aufgebaut ist mit
  - (R, S) ("R und S"),
  - $(R; S)$ ("R oder S"), oder
  - \+ R („nicht R").

Zum ersten Fall gehören einige eingebaute Prädikate, z.B. die 0-stelligen `true`, `fail`, `!`.

Ein Fakt oder Axiom ist eine Regel der Form *Kopf* `:- true.`

Ein Programm ist eine Folge von (*dann-wenn-*) Regeln,

$$\begin{array}{lcl} \text{Kopf}_1 & \text{:-} & \text{Rumpf}_1\,. \\ & \vdots & \\ \text{Kopf}_n & \text{:-} & \text{Rumpf}_n\,. \end{array}$$

Die Klauseln eines Programms P definieren die Bedeutung der Prädikate von P: das Prädikat p/n trifft auf die Tupel $(t_1, \ldots, t_n)$ von Termen zu, für die $p(t_1, \ldots, t_n)$ mit P beweisbar ist.

Bem. Funktionssymbole, d.h. Atome f in Termen $f(t_1, \ldots, t_n)$, haben nur als Prädikate f/n eine Bedeutung; es werden keine Funktionswerte (d.h. andere Terme) berechnet!

Beispiele

Wir geben das obige Programm zu $<$ in Prolog-Syntax wieder; da $<$ als Zahlvergleich eingebaut ist, benutzen wir ein Prädikat `less/2`. Neben zwei Fakten wird verlangt, daß `less/2` transitiv ist:

⟨*trans.pl*⟩≡

```
less(a,b).
less(b,d).
less(X,Z) :- less(X,Y), less(Y,Z).
```

Den „Datentyp“ *Genus* legen wir dadurch fest, daß mit einem Prädikat `genus/1` gesagt wird, welche Atome als Genuswerte angesehen werden:

⟨*genus.pl*⟩≡

```
% erlaubte Genus-Werte: mask, fem, neut
genus(mask).
genus(fem).
genus(neut).
```

Prolog starten/verlassen, Programme laden usw.

SWI-Prolog 6.6.4 ist in Linux-Distributionen verfügbar und auf den Rechnern des Instituts für Informatik installiert.

Man startet Prolog auf der Unix-Shell durch `pl` und wird mit `?-` zu Eingaben aufgefordert:

⟨*Aufruf von Prolog zur interaktiven Benutzung:*⟩≡

```
$ pl
Welcome to SWI-Prolog (... Version 6.6.4)
...
For help, use ?- help(Topic). or ?- apropos(Word).
?-
```

Dort kann man Programme laden und Anfragen stellen. Zum Verlassen dient das Prädikat `halt/0` (oder `^D`):

⟨*Beenden einer Prolog-Sitzung:*⟩≡

```
?- halt.
$
```

Ein Programm wird durch `consult/1` geladen:

⟨*Laden der Datei* `genus.pl`⟩≡

```
?- consult(genus).
% genus compiled 0.00 sec, 1,060 bytes
true.

?-
```

Prolog ergänzt das Suffix `.pl` oder `.pro` und lädt die Datei aus dem Verzeichnis, in dem man Prolog gestartet hat. Dateien aus anderen Verzeichnissen müssen mit Pfadnamen angegeben werden, wie `consult('Unterverzeichnis/genus.modified.pl')`

⟨*Laden mehrerer Dateien durch eine Liste von Dateinamen:*⟩≡

```
?- [genus,trans].
```

Anfragen

Nach dem Laden von Dateien kennt Prolog die darin angegebenen Axiome und Beweisregeln. (Beachte: wird ein Prädikat `p/n` in mehreren Dateien definiert, so kennt es nur die letzte Definition.)

Eine Anfrage ist, wie der Rumpf einer Regel,

- ein Term $q(s_1, \ldots, s_k)$ ist, wo q/k ein Prädikat ist, oder
- aus solchen Termen R und S aufgebaut mit
  - (R, S) ("R und S"),
  - $(R; S)$ ("R oder S"), oder
  - \+ R („nicht R“).

In den meisten Fällen stellt man einfache (atomare) Fragen, etwa

⟨*Einfache Anfrage:*⟩≡

```
?- genus(fem).  % Ist fem ein Genus-Wert?
true.
?- genus(akk).  % Ist akk ein Genus-Wert?
false.
```

⟨*Komplexe Ja/Nein-Fragen und ihre Antworten*⟩≡

```
?- genus(mask) , genus(fem). % Sind akk,fem Genus-Werte?
true.
?- genus(akk) ; genus(fem).  % Ist akk o. fem ein Genus?
true.
?- \+ genus(akk).    % Ist akk kein Genus-Wert?
true.
```

⟨*Frage mit Variablen und Beantwortung durch Terme*⟩≡

```
?- genus(X). % Welche X sind Genus-Werte?
X = mask ;  % 1.Anwort (mit ; weitere Antworten erbitten!
X = fem ;   % 2.Antwort
X = neut.   % 3.Antwort
```

⟨*Frage mit undefiniertem Prädikat*⟩≡

```
?- kasus(akk).
ERROR: toplevel: Undefined procedure: kasus/1
```

Antwort- bzw. Beweissuche

Sei das Programm P geladen und die Frage Q gestellt, etwa

$$
\begin{array}{lll}
 & P & \qquad\qquad Q \\
\text{Kopf}_1 & \text{:- Rumpf}_1\,. & \\
 & \vdots & ?-q(t1,..,tn). \\
\text{Kopf}_n & \text{:- Rumpf}_n\,. &
\end{array}
$$

Wie sucht Prolog die Antworten, d.h. beweist Q mit P, bzw. eine Spezialisierung von Q durch Belegen von Variablen?

1. Prolog sucht linear die erste Klausel, deren Kopf zur Anfrage „paßt“, d.h. wo durch eine Variablenbelegung S der Kopf und die Anfrage „identifiziert“ werden: $S(\text{Kopf}_i) = S(\text{Anfrage})$;
2. Dann versucht Prolog, den Rumpf der spezialisierten Klausel $S(\text{Kopf}_i)$:- $S(\text{Rumpf}_i)$. zu beweisen.

3. Beim Beweisen von $S(\text{Rumpf}_i)$ geht man analog vor, wobei die Variablenbelegung S i.a. weiter zu einem S' spezialisiert wird.
4. Falls nichts mehr zu beweisen ist, ist die entstandene Belegung S', angewendet auf die Variablen der Ausgangsfrage Q, die Antwort (oder true, falls Q keine Variablen hatte).
5. Ist Q eine komplexe Frage, so werden bei $Q = (R, S)$ beide Teile, bei $Q = (R; S)$ vorrangig R, dann S bewiesen, und bei $Q = \backslash+\ R$ wird versucht, R zu widerlegen, d.h. zu zeigen, daß R keinen Beweis hat.

Da i.a. mehrere Klauseln zu einer Frage $q(t_1, \ldots, t_n)$ „passen“ können, werden die Beweismöglichkeiten in einem Suchbaum dargestellt.

Ein Knoten des Baums ist mit einer Frage markiert, seine direkten Nachfolger mit den unmittelbaren Alternativen, deren Beantwortung eine Antwort auf die Frage wäre.

Auf der Kurseite gebe ich Ihnen ein Programm `graphviz_hl.pl`, das den Suchbaum zu einer Anfrage darstellt.[1]

Dazu ruft man `why(+Anfrage)` auf, wenn einen die Belegungen nicht interessieren, sonst `why2(+Anfrage)`. Oft muß man die Tiefe des Suchbaums einschränken: `why2(+Anfrage,+max.Baumtiefe)`.

Das geht natürlich nur bei sehr kleinen Programmen.

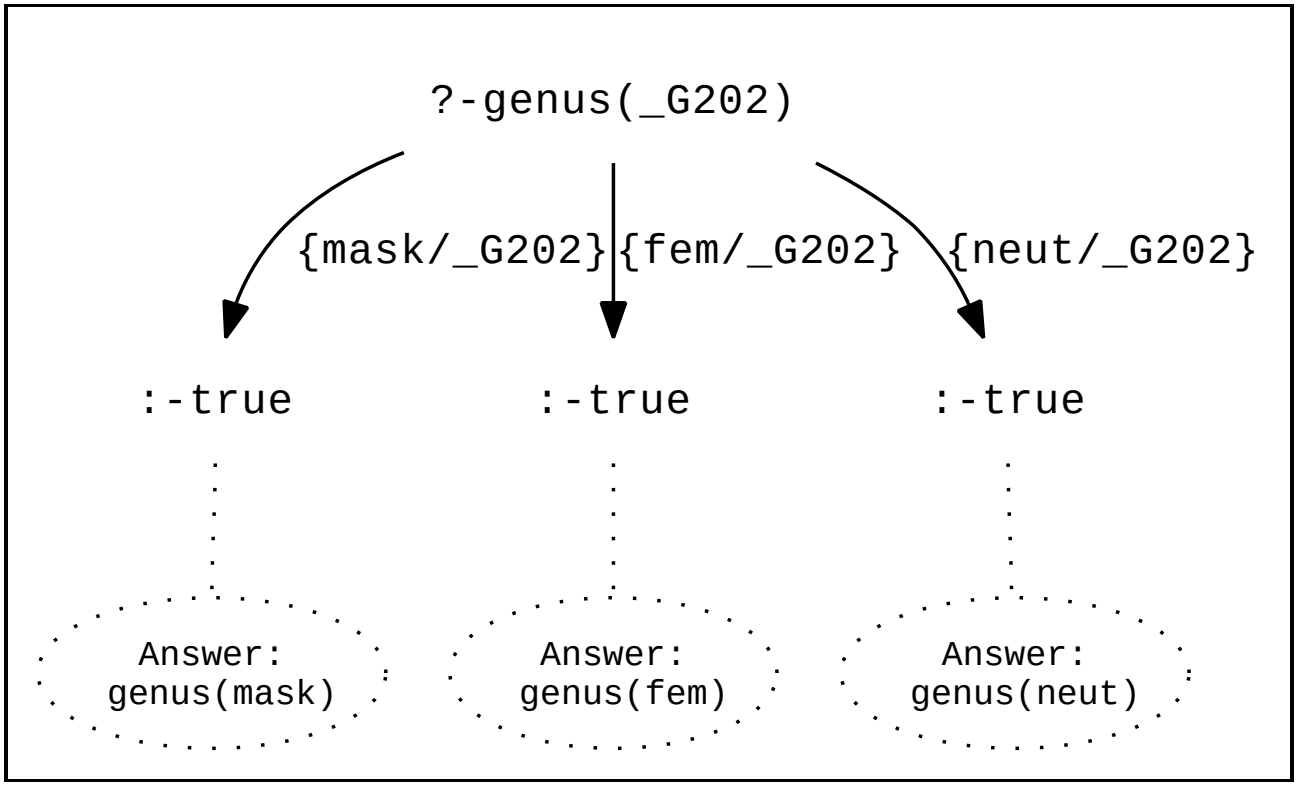
⟨*Anzeigen des Suchbaums*⟩≡

```
?- [graphviz\_hl, genus].
?- why((genus(akk);genus(fem))).
?- why2((genus(akk);genus(fem))). % why2/1 (+ Belegung)
?- why2(genus(X), 4).  % why2/2 mit max.Baumtiefe
```

[1] Es erfordert `dot` und `gv` (ghostview) (ggf. siehe Quelle in `graphviz_hl`).

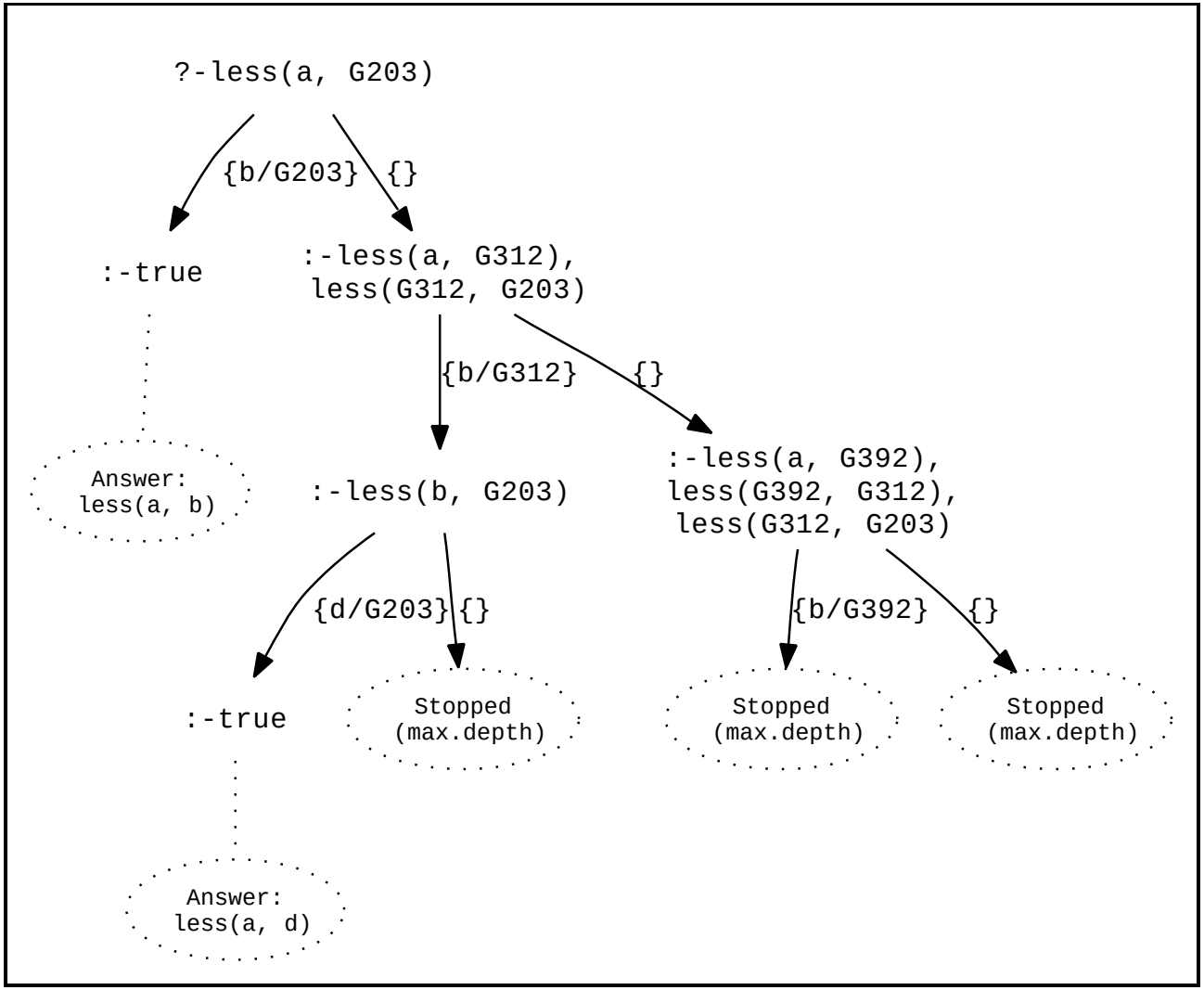
Beispiele für Suchbäume

Suchbaum für Beweise zur Anfrage `?- genus(G).` zum Programm `genus.pl` mit: `genus(mask). genus(fem). genus(neut).`



Die Variable in der Anfrage kann mit drei Werten so belegt werden, daß ein Faktum (Axiom) entsteht, d.h. nichts mehr zu beweisen ist.

Suchbaum für Beweise von `less(a,U)` zum Programm `trans.pl`



Definite Clause Grammars

Eine quantorenfreie logische Formel ist eine Hornformel (nach Alfred Horn, 1951) wenn sie in konjunktiver Normalform ist (d.h. eine Konjunktion von Disjunktionen aus atomaren und negiert atomaren Formeln), und in jeder Disjunktion alle bis auf ein Disjunktionsglied negiert sind.

Die Formel ist dann eine Konjunktion von Formeln der Form

$$\varphi_0 \vee \neg\varphi_1 \vee \ldots \vee \neg\varphi_n \quad \equiv \quad \varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi_0$$

mit atomaren Formeln φ_i, die den dann-wenn-Klauseln

$$\varphi_0 \text{ :- } \varphi_1, \ldots, \varphi_n.$$

von Prolog entsprechen. (Solche Klauseln heißen definit, weil in φ_0 keine Alternative möglich ist.)

Ein Prolog-Programm ist daher logisch gesehen eine Hornformel.

Eine Definite Clause Grammar oder Hornformelgrammatik ist, grob gesagt, eine Grammatik, deren Kategorien durch solche Formeln definiert werden.

Eine grammatische Kategorie wie *Satz* oder *Nominalphrase* ist ein *Prädikat über Wortfolgen*, sodaß man Aussagen wie

s(der alte Baum fällt), oder: np(der alte Baum),

bilden kann. Für das Parsen ist es sinnvoller, eine grammatische Kategorie als *2-stellige Relation zwischen Wortfolgen* zu verstehen:

$$np(\overbrace{\underbrace{\textit{der alte Baum}}_{\textit{Nominalphrase}}\ \textit{fällt um}}^{\textit{Eingabe } I},\ \overbrace{\textit{fällt um}}^{\textit{Rest } J})$$

Das 2-stellige und das 1-stelligen Prädikat hängen so zusammen:

$$np(I, J) \iff np(I - J), \qquad I - J := I \text{ ohne Suffix } J$$

Der praktische Vorteil, die Kategorien C als 2-stellige Relationen zwischen Wortfolgen zu verstehen, besteht darin, daß man

- zu gegebener Eingabe I am Anfang einen Ausdruck der Kategorie C sucht, entfernt und
- den unbenutzten Rest J mit anderen Kategorien analysiert.

Eine kontextfreie Grammatikregel wie `(s -> np vp)` soll besagen, daß eine Wortfolge I ein Satz ist, wenn sie die Verkettung $K + L$ einer Nominalphrase K mit einer Verbalphrase L ist:

⟨*Satzregel mit Eigenschaften von Wortfolgen formuliert:*⟩≡

```
s(I) :- np(K), vp(J), append(K,J,I). % K+J = I
```

Stellt man das Präfix K als Differenzliste $I - J$ von Gesamtliste I und Restliste J dar, so spart man sich die Verkettung:

⟨*Satzregel mit Relationen zwischen Wortfolgen formuliert:*⟩≡

```
s(I,[]) :- np(I,J), vp(J,[]). % I=I-[], K=I-J, J=J-[]
```

In der Programmiersprache Prolog kann man kontextfreie Grammatiken, erweitert um Merkmale, leicht definieren:

1. Kategorien: `<Kat>` ::= Prolog-Terme (außer Variable, ...)
2. Startsymbole: `startsymbol(<Kat>) :- <Bedingung>.`
3. Regeln:

```
<Regel>  ::=  <Kat> --> <RS>.
    <RS>  ::=  <Kat>                        Nonterminal
            |  (<RS> , <RS>)                Sequenz
            |  (<RS> ; <RS>)                Alternative
            |  [<Atom>,...,<Atom>]          Terminale
            |  {<Prolog>}                   Nebenbedingung
```

4. Hilfsprädikate: durch Prolog-Klauseln definierbar

Abkürzungen: `X,Y,Z` für `(X,(Y,Z))` `X;Y;Z` für `(X;(Y;Z))`

Beispiel einer DCG

⟨*DCGparser/testgrammatik.pl*⟩≡

```
startsymbol(s(Tmp)) :- tempus(Tmp).
startsymbol(np(Kas)) :- kasus(Kas).

kasus(nom).        kasus(akk).
tempus(praes).     tempus(praet).

s(Tmp) --> np(nom), tv(Tmp), np(akk), {tempus(Tmp)}.
np(Kas) --> art(Kas), a(Kas), n(Kas), {kasus(Kas)}.
art(nom) --> ['Der'] ; [der].
art(akk) --> ['Den'] ; [den].
a(nom) --> [alte] ; [junge].
a(akk) --> [alten] ; [jungen].
n(Kas) --> (['Mann'] ; ['Baum']), {kasus(Kas)}.
tv(praes) --> [faellt] ; [fällt].
tv(praet) --> [faellte] ; [fällte].
```

Beim Einlesen einer DCG in Prolog werden

- die *n*-stelligen Kategorien in n+2-stellige Prolog-Prädikate übersetzt, z.B. `np/0` in `np(+Eingabeliste,?Restliste)`,
- und die DCG-Regeln in Prolog-Klauseln übersetzt

Die DCG-Regel `s --> np, vp.` hat 0-stellige Kategorien und wird in die Prolog-Klausel mit $0 + 2 = 2$-stelligen Prädikaten übersetzt:

⟨*Übersetzung der DCG-Regel durch Prolog:*⟩≡

```
s(A, C) :-
        np(A, B),
        vp(B, C).
```

Aus der DCG wird ein Erkenner, der die grammatische Korrektheit einer Eingabe feststellen kann, aber keine Syntaxanalyse ausgibt.

Um einen Parser zu bekommen, haben wir eine *andere* Übersetzung von DCG-Regeln in Prolog-Klauseln in `DCGparser/parser.pl` programmiert, die

- die *n*-stelligen Kategorien in n+3-stellige Prolog-Prädikate, z.B. `np/0` in `np(-Syntaxbaum,+Eingabeliste,?Rest)`,
- und die DCG-Regeln in Prolog-Klauseln mit Baumausgabe übersetzt.

Wird *zuerst* `DCGparser/parser.pl` und *dann* die DCG-Regel `s -> np, vp.` geladen, so entsteht die Klausel

⟨*Übersetzung der DCG-Regel mit* `parser.pl`*:*⟩≡

```
s([s, A, C], B, E) :-
        np(A, B, D),
        vp(C, D, E).
```

Hier ist `[s,A,C]` der Baum für den Satz `B-E`, der aus der Wurzel `s` und Teilbäumen `A` und `C` besteht, die man beim Parsen der Nominalphrase `B-D` und der Verbalphrase `D-E` bekommt.

Das Programm `parse/0` aus `DCGparser/parser.pl`

- erlaubt das Eintippen der Eingabe statt einer Anfrage
  `?- np(Baum,Liste der Eingabewörter,Restliste)`
- gibt den Syntaxbaum durch einen formatierten Text statt durch eine Prolog-Liste

  `[Kategorie, Teilbaum1,...,TeilbaumN]`

  aus.

Das Programm `parsed/0` aus `DCGparser/parser.pl` gibt den Baum in einem eigenen Fenster als Graphik aus, braucht dazu aber die Hilfsprolgramme `graphviz/dot` und `gv` (ghostview).

Bem. Wenn man verstehen will, wie Prolog die Syntaxanalyse vornimmt, kann man (bei einer kleinen Grammatik) das Hilfsprogramm `graphviz_hl.pl` laden und die Anfrage zum Parsen in `why2(+Anfrage,+Tiefe des Suchbaums)` stellen:

⟨*Suchen der Syntaxanalyse:*⟩≡

```
?- ['DCGparser/parser.pl'].
?- ['DCGparser/testgrammatik.pl'].
?- [graphviz_hl].
?- why2(np(Kasus,Baum,[den,alten,'Baum'],Rest),15).
true.
```

Da der Suchbaum relativ groß ist, könnte man das einfacher auch *ohne* die Baumausgabe probieren, d.h. indem man nur `testgrammatik` und `graphviz_hl` lädt und nur

⟨*Suchbaum für das Erkennen*⟩≡

```
?- why2(np(Kasus,[den,alten,'Baum'],Rest),15).
```

fragt.