8 Grundlagen der logischen Programmierung: deklarative Programmierung

Logisches Programmieren (z.B. Prolog) hat als theoretische Basis die Prädikatenlogik erster Stufe, und als Ausführung der Programme eine Variante der Resolution.

Programme in logischen Programmierens verwenden spezielle Klauseln, sogenannte Horn-Klauseln.

Definition 8.1.

- *Eine* Hornklausel *ist eine Klausel mit maximal einem positiven Literal*[7].
- *Eine* definite Klausel *ist eine Klausel mit genau einem positiven Literal. D.h. die Klausel ist von der Form:* $\neg A \vee B_1 \vee \ldots \vee B_m$ *oder in der in der logischen Programmierung verwendeten Notation:*

  $$A \Leftarrow B_1, \ldots, B_m \quad \mathit{bzw.} \quad A \text{ :- } B_1, \ldots, B_m$$

  Man nennt A *den Kopf,* $\{B_1, \ldots, B_m\}$ *den Rumpf der Klausel.*
- *Eine Klausel ohne positive Literale nennt man* definites Ziel *(*Anfrage,Query,goal*). Notation:*

  $$\Leftarrow B_1, \ldots, B_n$$

  Die B_i *werden* Unterziele (Subgoals) *genannt.*
- *Eine* Unit-Klausel *(oder* Fakt*) ist eine definite Klausel mit leerem Rumpf. Notation:* $A \Leftarrow$.
- *Ein* definites Programm *ist eine Menge von definiten Klauseln.*
- *Die Menge aller Klauseln, deren Kopfliteral das Prädikat* Q *hat, nennen wir* Definition von Q.

Ein definites Programm entspricht einer Und-Verknüpfung aller definiten Klauseln. Beachte, daß jede Hornklausel entweder eine definite Klausel oder ein definites Ziel ist.

Ein definites Ziel kann man in PL_1 so hinschreiben:

$$\forall x_1, \ldots, x_n : \neg B_1 \vee \ldots \vee \neg B_n$$

Dies ist äquivalent zu:

$$\neg \exists x_1, \ldots, x_n : B_1 \wedge \ldots \wedge B_n$$

D.h. daß man eine Anfrage als eine existentiell quantifizierte Aussage sehen kann:

$$\exists x_1, \ldots, x_n : B_1 \wedge \ldots \wedge B_n$$

deren Negation zum Programm hinzugefügt wird. Die Herleitung besteht in der Erzeugung eines Widerspruchs (der leeren Klausel $\Box$). Die Herleitung des Widerspruchs kann bei Hornklauseln immer so gemacht werden, daß man die existierenden Objekte erzeugt.

[7] Ein positives Literal ist ein Atom; ein negatives Literal ist ein negiertes Atom

8.1 SLD-Resolution

SLD-Resolution ist die prozedurale Semantik von definiten Programmen. D.h. SLD-Resolution definiert die Ausführung von logischen Programmen:

S: selection function
L: lineare Resolution
D: definite clauses

Zunächst erklären wir die SLD-Resolution in ihrer allgemeinen (nicht-deterministischen) Form:

Definition 8.2. *Sei* $G = \quad \Leftarrow A_1, \ldots, A_m, \ldots, A_k$ *ein definites Ziel und* $C = A \Leftarrow B_1, \ldots, B_q$ *eine definite Klausel. Dann kann man aus* G *und* C *ein neues Ziel* G' *wie folgt mit Resolution* herleiten*:*

1. A_m *ist das selektierte Atom des definiten Ziels* G*.*
2. θ *ist ein allgemeinster Unifikator von* A_m *und* A*, dem Kopf von* C*.*
3. G' *ist das neue Ziel:* $\theta(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)$*.*

Man sieht: G' ist eine Resolvente von G und C. Die Ableitungsrelation sei bezeichnet durch $G \rightarrow_{P,C,m} G'$, wobei man die genaue Kennzeichnung P, C, m auch weglassen kann, wenn diese aus dem Kontext hervorgehen.

Definition 8.3. *Sei* P *ein definites Programm und und* G *ein definites Ziel. Eine* SLD-Ableitung *von* $P \cup \{G\}$ *ist eine Folge*

$$G \rightarrow_{C_1,m_1} G_1 \rightarrow_{C_2,m_2} G_2 \ldots$$

von SLD-Schritten, wobei C_i *jeweils eine Variante einer Klausel aus* P *ist mit neuen Variablen.*

Die SLD-Ableitung ist eine SLD-Widerlegung*, wenn sie mit einer leeren Klausel endet.*

Im Sinne der linearen Resolution nennt man die Klauseln C_i *die Eingabeklauseln.*

Weitere Sprechweisen:

erfolgreiche SLD-Ableitung: Wenn es eine SLD-Widerlegung ist.
fehlgeschlagene SLD-Ableitung: Wenn diese nicht fortsetzbar ist.
unendliche SLD-Ableitung

Definition 8.4. *Sei* P *ein definites Programm und* G *ein definites Ziel.*

- *Eine* korrekte Antwort *ist eine Substitution* θ*, so daß* $P \models \theta(\neg G)$ *gilt.*
- *Eine* berechnete Antwort θ *für* $P \cup \{G\}$ *ist eine Substitution, die man durch Einschränkung von* $\theta_n \circ \ldots \circ \theta_1$ *auf die Variablen von* G *erhält, wobei* $\theta_1 \ldots \theta_n$ *die allgemeinsten Unifikatoren (in dieser Reihenfolge) zu einer SLD-Widerlegung von* G *sind.*

Satz 8.5. Soundness der SLD-Resolution
Sei P *ein definites Programm und* G *ein definites Ziel. Dann ist jede berechnete Antwort* θ *auch korrekt. D.h.* $P \models \theta(\neg G)$

8.2 Vollständigkeit der SLD-Resolution

Wir werden verschiedene Vollständigkeitsaussagen für die SLD-Resolution formulieren.

Satz 8.6. *(Widerlegungsvollständigkeit)*
Sei P ein definites Programm und G ein definites Ziel. Wenn $P \cup \{G\}$ unerfüllbar ist, dann gibt es eine SLD-Widerlegung von $P \cup \{G\}$.

Es gilt der folgende stärkere Satz über die Vollständigkeit der berechneten Antworten, nämlich daß es zu jeder Antwortsubstitution eine berechnete Antwort gibt, die allgemeiner ist.

Satz 8.7. (Vollständigkeit der SLD-Resolution)
Sei P ein definites Programm und G ein definites Ziel. Zu jeder korrekten Antwort θ gibt es eine berechnete Antwort σ für $P \cup \{G\}$ und eine Substitution γ so daß für alle Variablen $x \in FV(G)$: $\gamma\sigma(x) = \theta(x)$.

8.3 Strategien zur Berechnung von Antworten

Definition 8.8. *Der folgende Algorithmus berechnet alle Antworten mit einer breadth-first Strategie:*

1. *Gegeben ein Ziel $\Leftarrow A_1, \ldots, A_n$:*
   (a) *Probiere alle Möglichkeiten aus, ein Unterziel A aus $A_1, \ldots, A_n$ auszuwählen*
   (b) *Probiere alle Möglichkeiten aus, eine Resolution von A mit einem Kopf einer Programmklausel durchzuführen.*
2. *Erzeuge neues Ziel B: Löschen von A, Instanziieren des restlichen Ziels, Hinzufügen des Rumpfs der Klausel.*
3. *Wenn $B = \Box$, dann gebe die Antwort aus.*
   Sonst: mache weiter mit 1 mit dem Ziel B.

Dieser Algorithmus hat zwei Verzweigungspunkte pro Resolutionsschritt:

- Die Auswahl eines Atoms
- Die Auswahl einer Klausel

Es stellt sich heraus, daß bei der Auswahl des Atoms die restlichen Alternativen nicht betrachtet werden müssen.

Aussage 8.9. *Vertauscht man in einer SLD-Widerlegung die Abarbeitung zweier Atome in einem Ziel, so sind die zugehörigen Substitutionen bis auf Variablenumbenennung gleich und die Widerlegung hat die gleiche Länge.*

Weiterhin: die Suchstrategie, die irgendein Atom auswählt, dann alle Klauseln durchprobiert, usw. ist vollständig bzgl der Antworten.

D.h.man kann leicht zu lösende Atome zuerst auswählen und schwerer zu lösende zurückstellen.

Es kann aber sein, daß bei günstiger Auswahl nur endlich viele Alternativen ausprobiert werden müssen, aber bei ungünstiger Auswahl evtl. eine unendliche Ableitung ohne Lösungen mitbetrachtet werden muß. Dies betrifft nicht die *Reihenfolge, in der Klauseln ausgewählt werden*, diese ist nach wie vor nichtdeterministisch.

8.4 Implementierung logischer Programmiersprachen: Prolog

In Implementierung wird i.a. die Suche durch verschiedene Vereinfachungen und Festlegungen deterministisch gemacht:

- Die Anfrage wird als geordnete Liste von Literalen implementiert. Es wird immer das erste Unterziel zuerst bearbeitet.
- Die Programmklauseln werden in der Reihenfolge abgesucht, in der sie im Programm stehen.
- Bei Ausführung eines SLD-Resolutionsschritts wird der Rumpf an die Stelle des Unterziels gesetzt, und zwar in der Reihenfolge der Literale in der Programmklausel.

Weitere Veränderungen gegenüber der theoretischen Fundierung sind:

- Der occurs-check wird i.a. nicht durchgeführt.
- Es gibt extra Operatoren zum Beeinflussen des Backtracking (*Cut*), der u.a. bewirken kann daßfür ein Unterziel statt vielen Lösungen maximal eine berechnet wird.
- Assert/Retract: Damit können Programmklauseln zum Programm hinzugefügt bzw. gelöscht werden.
- Negation: Negation wird definiert als Fehlschlagen der Suche.
- Es wird z.T. Typinformation zu Programmen hinzugefügt.
- Argumente von Prädikaten kann man mit dem Zusatz I bzw. O versehen. Diese Angaben bedeuten, daßdas jeweilige Argument nur als Eingabe bzw. nur als Ausgabe verwendet wird. z.B.
- Zusätzlich kann man ein Prädikat als Funktion definieren, wenn die ersten Argumente die Eingabe sind, das letzte Argument die Ausgabe, wobei die Funktion noch zusätzlich als deterministisch (d.h. als mathematische Funktion) deklariert werden muß.

In Implementierungen Folgende Tabelle vergleicht die theoretischen Eigenschaften mit denen implementierter logischer Programmiersprachen.

Pures Prolog Definite Programme	nichtpures Prolog Cut, Negation, Klauselreihenfolge fest	nichtpur, ohne occurs-check
SLD: ist vollständig	SLD: unvollständig	SLD: i.a. nicht korrekt

Gibt man eine Reihenfolge der Klauseln vor, wie in einer Prolog-implementierung, dann ist die SLD-Resolution nicht mehr vollständig: Manche Lösungen werden gefunden und aufgezählt, aber:

- es kann ein Schleife auftreten ohne ein weitere Antwort, obwohl es noch welche gibt.
- Es werden unendlich viele Antworten aufgezählt, aber nicht alle korrekten Antworten werden abgedeckt.

Es folgt auch, daß die Reihenfolge der Literale im Rumpf keinen Einfluß auf die Vollständigkeit hat.

Ein Beispiel für die Unvollständigkeit der festgelegten Reihenfolge der Klauseln:

```
P(a,b).
P(c,b).
P(X,Y) :- P(Y,X).
P(X,Z) :- P(X,Y), P(Y,Z).
```

Die symmetrische-transitive Hülle kann nicht berechnet werden, da das Programm in Schleifen gerät. Aber eine SLD-Widerlegung existiert, da diese nichtdeterministisch vorgehen darf, und insbesondere sich eine Programmklausel aussuchen darf.

8.5 SLD-Bäume

Aufgrund der obigen Vollständigkeitsaussage können wir SLD-Bäume definieren, die den ganzen Suchraum repräsentieren für ein Ziel, gegeben ein definites Programm.

Definition 8.10. *Gegeben ein definites Programm P, und ein definites Ziel G: Ein* SLD-Baum *für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:*

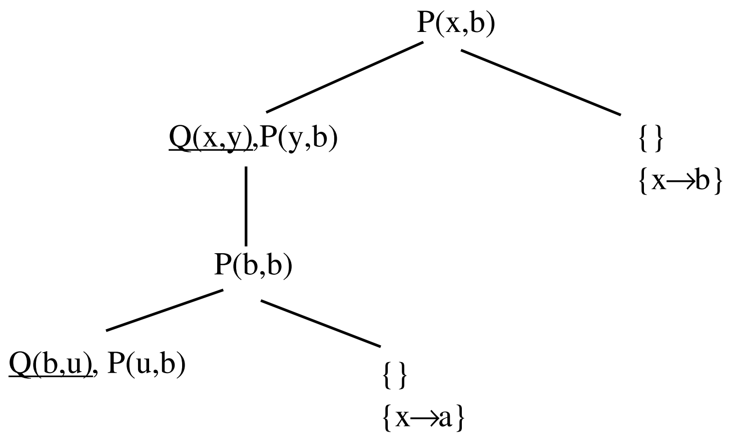
1. *Jeder Knoten ist markiert mit einem definiten Ziel*
2. *die Wurzel ist markiert mit G*
3. *In jedem Knoten mit nichtleerem Ziel wird ein Atom A des Ziels ausgewählt. die Söhne dieses Knoten sind dann die möglichen Ziele nach genau einem SLD-Resolutionsschritt mit einer Regel in P.*
4. *Knoten, die mit der leeren Klausel markiert sind, sind Blätter.*

5. *Ein Blatt ist entweder mit dem leeren Ziel markiert, oder es gibt von dem Blatt aus keine SLD-Resolution.*

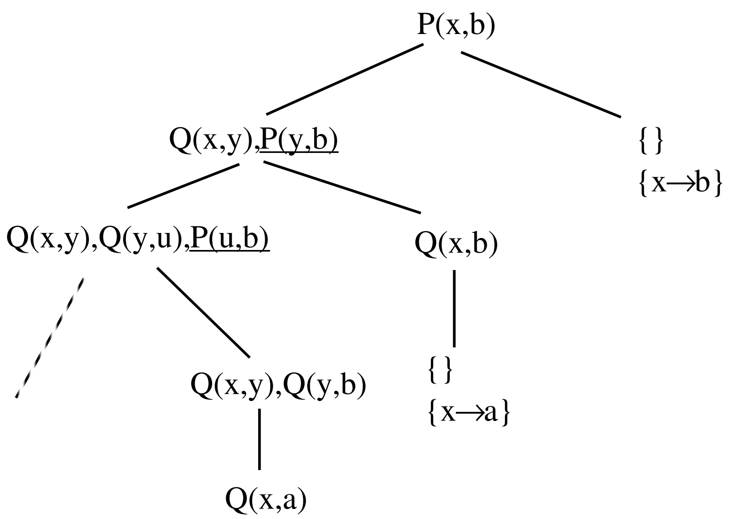
Äste entsprechen SLD-Ableitungen. Erfolgreiche Widerlegungen sind Erfolgspfade, unendliche SLD-Ableitungen entsprechen unendliche Pfaden. Fehlschläge entsprechen Pfaden zu Blättern, die mit einem nichtleeren Ziel markiert sind.

Beispiel 8.11. Programm:

```
P(X,Z) :-  Q(X,Y), P(Y,Z).
P(X,X).
Q(a,b).
```



Nimmt man eine andere Auswahl, dann kann der Baum unendlich werden:



An diesen SLD-Bäumen kann man (im Prinzip) den ganzen Suchraum ablesen.

8.6 Beispielprogramme und Ausführung

Programm 1:

```
likes (maria, essen).
likes(maria,wein).
likes(peter, maria).
likes(peter,wein).
```

Einfache Anfragen:

```
>   ?- likes(peter,maria)
       yes
```

Konjunktive Anfragen (mit logischem und verknüpfte)

```
>  ?- likes(peter,maria), likes(maria,peter)
   no
```

Zuerst wird das erste Unterziel bearbeitet: Antwort:. ja, Dann zweite: nein.
Konjunktive Anfragen mit Instanziierung.

```
>  ?- likes(peter, X), likes(maria,X)
```

Abarbeitung: 1. X = maria, likes(maria,maria): no

2. X = wein: likes(maria,wein). yes

```
X = wein
```

Backtracking ist notwendig (Zurücksetzen) zum Finden weiterer Lösungen.

Programm 2:

```
vater(peter,maria).
mutter(susanne,maria).
vater(karl, peter).
mutter(elisabeth, peter).
...

vorfahr(X,Y) :- vater(X,Y).
vorfahr(X,Y) :- mutter(X,Y).
vorfahr(X,Z) :- vorfahr(X,Y),vorfahr(Y,Z).
```

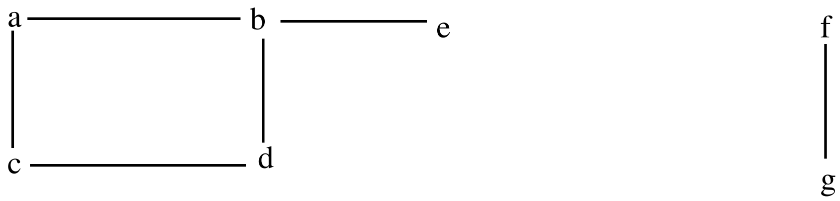
Anfragen:

```
?-  vorfahr(elisabeth,maria)
```

Abarbeitung :

```
 1. vorfahr(elisabeth,maria)
  1.1 vater(elisabeth,maria). nein
  1.2 mutter(elisabeth,maria): nein
  1.3 vorfahr(elisabeth, X0), vorfahr(X0,maria)
    1.3.1  vater(elisabeth, X0), vorfahr(X0,maria): nein
    1.3.2 mutter(elisabeth, X0), vorfahr(X0,maria):  X0 = peter
     1.3.2  vorfahr(peter,maria):
      1.3.2.1  vater(peter,maria): ja
  usw.
```

Beispiel 8.12. **ungerichteter Graph.**




```
    kante(a,b).
    kante(a,c).
    kante(c,d).
    kante(d,b).
    kante(b,e).
    kante(f,g).

    verbunden(X,Y) :- kante(X,Y).
    verbunden(X,Y) :- kante(Y,X).
    verbunden(X,Z) := verbunden(X,Y),verbunden(Y,Z).

?- verbunden(c,e)

     verbunden(c,Y1), verbunden(Y1,e)
    ...
     verbunden(c,a), verbunden(a,e)

     verbunden(a,Y2), verbunden(Y2,e)

     verbunden(a,b), verbunden(b,e)
     yes.
```

Übungsaufgabe 8.13. was passiert bei folgender Eingabe?

```
?- verbunden(e,f)
```

8.7 Syntaxkonventionen von Prolog

Namen können aus Großbuchstaben, Kleinbuchstaben, Ziffern und _ (Unterstrich) gebildet werden, oder nur aus Sonderzeichen.

Konstanten sind Namen, die mit Kleinbuchstaben beginnen Variablen sind Namen die mit Großbuchstaben beginnen, oder mit einem Unterstrich : _. Der Unterstrich alleine _ ist der Name einer anonymen Variablen (wildcard, joker, Leerstelle)

zusammengesetzte Terme (komplexe Terme) haben die Form

$$Functor(component_1, \ldots, component_n)$$

Die Stelligkeit des Funktors ist nicht variabel. Verwendet man den gleichen Namen mit verschiedener Stelligkeit, so werden diese Vorkommen als verschiedene Objekte interpretiert

Funktoren, die auf oberster Ebene stehen, nennt man auch Prädikate. Z.B. `besitzt(peter, buch(lloyd, prolog))`.

8.8 Darstellung arithmetischer Ausdrücke:

Die Zeichen $+, -, *, /$ sind erlaubt.

Ausdrücke der Form $a * b + c$ sind lesbare Abkürzungen für den Term $+(*(a, b), c)$. Hier kann zur Darstellung abhängig von der Prolog-Implementierung Infix, Postfix, Präfix, Assoziativität, Priorität benutzt werden.

Zahlen sind erlaubt und in Prolog bekannt. Der Zahlbereich ist abhängig von der Implementierung.

Spezialprädikat $=$ (Gleichheit) kann definiert werden durch

```
X = X.
```

Z.B.

```
?- besitzt(peter, X) = besitzt(peter,buch(lloyd,prolog))
   yes; X = buch(lloyd,prolog)
```

Diese Operation der Berechnung der Instanzen von Variablen wird mittels Unifikation durchgeführt.

arithmetische Vergleiche:

```
=, /=, <, >, =<, >= , =\=, =:=
```

sind möglich zwischen instantiierten Werten.

D.h.: wenn exp1 op exp2 ausgewertet wird, muss exp1 und exp2 zu einer Zahl auswertbar sein. Beachte:

$3 + 4 = 7$ ergibt: no

$3 + 4 =:= 7$ ergibt yes

D.h., es gibt einen Unterschied zwischen Term und Wert.

Will man einer Variablen einen Wert zuweisen, so benötigt man das “is“ Prädikat:

$X \; is \; 3+5$ danach: $X \mapsto 8$

$X \;=\; 3+5$ danach $X \mapsto$ “$3+5$“

Relationale Addition und Multiplikation kann man verwenden bzw. definieren:

```
times(X,Y,Z)
plus(X,Y,Z)
```

Diese Prädikate kann man vorw—ärts und rückwärts verwenden.

```
?- plus(X,Y,5)
     X = 1,Y = 4
     X = 2, Y = 3
     ...

?- plus(2,3,X)

    X = 5
```

8.9 Datenstrukturen und Unifikation in Prolog

Intuition: Die Bäume werden gleichgemacht durch Instantiierung der Variablen.

$$k(X,a) = k(b,Y) \text{ ergibt } X = b, Y = a$$

Auch Unterstrukturen können Wert von Variablen sein:

$$k(X,h(a,Z)) = k(b,Y) \text{ ergibt } X = b, Y = h(a,Z)$$

Etwas schwierigere Fälle sind:

$$k(X,h(a,Z),X) = k(h(Y,b),U,U)$$

ergibt zunächst:

$$X = h(Y,b), U = h(a,Z), X = U$$

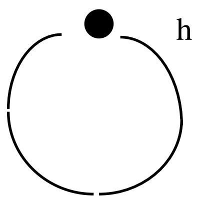
Endergebnis:

$$X = h(a,b), U = h(a,b), Y = a, Z = b$$

Behandlung des **occurs-check**:

$$k(X,X) = k(Y,h(Y))$$

erfordert Unifikation von $X = h(X)$.
Mit unendlichen Bäume als Termen ist die Unifikation möglich: $X = h(X)$ kann gleichgemacht werden durch: $X = h(h(h(\ldots)))$. Dieses Objekt kann als gerichteter Graph dargestellt werden:



Prolog-Implementierungen vermeiden den occurs-check. Teilweise wird er durch statische Analyse eliminiert. Bzw. wird ganz abgeschaltet. In diesem Falle nimmt man in Kauf, daß das Programm bzgl der PL_1-Semantik inkorrekt wird.

8.10 Listen:

Diese können in logischen Programmiersprachen direkt dargestellt werden. Man benutzt im allgemeinen eine abkürzende Syntax:

$[1, 2, 3]$ entspricht `cons(1,cons(2,cons(3,NIL)))`

[] entspricht NIL

Beispiel für den Element-Test:

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Terminiert `member` in allen möglichen Situationen?

- `member(s,t)`: wenn t keine Variablen enthält. dann wird t' beim nächsten mal kleiner.
- `member(Y,Y)`: Die erste Klausel unifiziert nicht, wenn occurs-check eingeschaltet ist, (aber unifiziert, wenn occurs-check aus ist: Antwort: ja) Die zweite Klausel ergibt:

```
member(X_1,[\_|Y_1]) :-  member(X_1,Y_1).
X_1 = Y = [\_|Y_1]
```

Die neue Anfrage ist:

```
member([\_|Y_1] ,Y_1)
```

Nächste Unifikation ebenfalls mit zweiter Klausel

```
member(X_2,[\_|Y_2]) :-  member(X_2,Y_2)
X_2 = [\_|Y_1] , Y_1 = [\_|Y_2]
```

usw. Man sieht: das terminiert nicht.

```
islist([_|X]) :- islist(X).
islist([]).
```

Diese Funktion terminiert ebenfalls für Listen ohne Variablen aber nicht für `islist(X)`.
Eine Umstellung der Klauseln ergibt:

```
islist([]).
slist([_|X]) :- islist(X).
```

Dieses Programm terminiert für die Anfrage `islist(X)`.
Antwort: $X = []$, falls die Eingabe eine Liste ist. Antwort ist dann ja.

```
laenge([],0).
laenge([_|X],N) :- laenge(X,N_1), N is N_1 + 1.
```

Damit kann man folgende Beispiele rechnen:

```
 ?-  laenge([1,2,3], N).
    N = 3.
```

Die Anfrage

```
?- laenge(X,2).
```

terminiert in manchen Implementierungen nicht: Wenn `is` auch rückwärts funktioniert, dann ergibt diese Anfrage als Antwort $X = [\_1, \_2]$.
Beispiel sortierte Listen von Zahlen:

```
sortiert([]).
sortiert([X]).
sortiert([X,Y|Z]) :- X <= Y, sortiert([Y|Z]).

sortiert_einfuegen(X,[],[X]).
sortiert_einfuegen(X,[Y|Z], [X,Y|Z]) :-  X =< Y.
sortiert_einfuegen (X,[Y|Z], [Y|U]) :-  Y < X, sortiert_einfuegen(X,Z,U).

ins_sortiere(X,X) :-  sortiert(X).
ins_sortiere([X|Y], Z) :- ins_sortiere(Y,U), sortiert_einfuegen(X,U,Z).
```

Programmierung von `append:` in Prolog:

```
append([],X,X).
append([X|Y],U,[X|Z]) :- append(Y,U,Z).
```

8.11 Parsen in Prolog

Eine häufige Anwendung der speziellen Abarbeitungsstrategie von Prolog ist die Verwendung als Implementierungssprache für Parser zu gegebenen Grammatiken, insbesondere für natürliche Sprachen [?,?,?]. Dies ist auch historisch gesehen die erste Anwendung gewesen. Prolog verwendet als Suchstrategie Tiefensuche mit Backtracking. Dies kann direkt für einen rekursiv absteigenden Parser verwendet werden, wobei man nicht nur kontextfreie Grammatiken verwenden kann, sondern auch erweiterte Grammatiken, z.B. für (schriftliche) Eingabe in natürlicher Sprache. Diese haben als Erweiterung, daß bestimmte Attribute der Terminale testbar sind, z.B. Geschlecht, Fall, Zeit, usw. und die Akzeptanz einer Phrase steuern können.

Kontextfreie Grammatiken für einfache englische Sprache

Es folgt eine Tabelle der *syntaktischen Kategorien:*

Det	Determiner (Artikel)	the, a some
N	Noun (Nomen)	table, computer, John
V	verb	(writes, eats, having)
ADJ	adjectives	(big, fast)
ADV	adverbs	(very, slowly, yesterday)
AUX	auxiliaries (Hilfsverben)	(is, do, has will)
CON	conjunctions	(and, or)
PREP	prepositions	(to, on, with)
PRON	Pronouns Pronomen	(he, who, which)

Weitere Bezeichnungen (Subkategorisierung und Komponenten):

S	Sentence	(Satz)
PN	proper noun	(„a .... “ ist verboten)
IV	intransitive verb	hat kein Objekt
TV	transitives verb	hat Objekt.

Komponenten eines Satzes (Phrasen, Sätze):

S	(Sentence) Satz	
NP	Nounphrase Nominalphrase	das dicke Buch
VP	Verbphrase	schreibe ein Buch
PP	Präpositionalphrase	mit einem Fernglas
ADJP	adjective phrase	größer als man erwartet
ADVP	adverbial phrase	(yesterday evening, gestern abend)
OptRel	relative clause	

Die (erste Annäherung an eine) Beschreibung der Grammatik einer natürlichen Sprache erfolgt mit *Phrasenstrukturregeln*. Diese können mit CFGs zur Konstruktion sowie zur Analyse von Sätzen bzw. Phrasen verwendet werden. Diese Grammatik ist natürlich abhängig von der beschriebenen Sprache (Deutsch, Englisch, Französisch, Russisch, usw.).

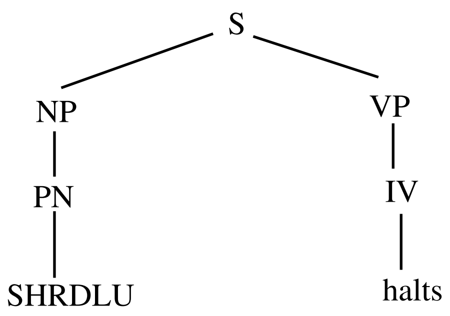
Eine kontextfreie Grammatik (CFG) zum Erzeugen einfacher englischer Sätze kann man wie folgt schreiben:

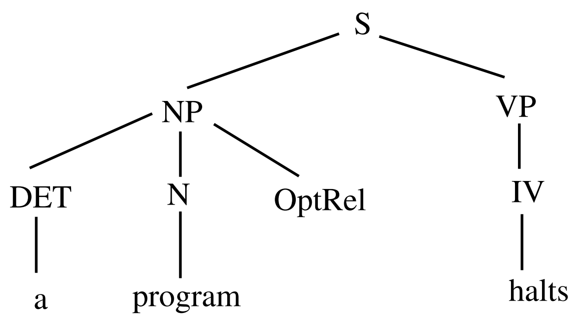
S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N OptRel
NP	$\rightarrow$	PN
Optrel	$\rightarrow$	ε
Optrel	$\rightarrow$	that VP
VP	$\rightarrow$	TV NP
VP	$\rightarrow$	IV

Lexikoneinträge:

PN	$\rightarrow$	terry
PN	$\rightarrow$	shrdlu
Det	$\rightarrow$	a
N	$\rightarrow$	program
IV	$\rightarrow$	halts
TV	$\rightarrow$	writes

Beispiel 8.14.





Der Anfang des Prologprogramms, das eine als Liste von Worten gegebenen Satz parst (erkennt), sieht so aus:

```
istsatz(Ein)        :-  s(Ein,[]).
s(Ein,Rest)         :-  np(Ein,Rest1),vp(Rest1, Rest).
np(Ein,Rest)        :-  det(Ein,Rest1), n(Rest1,Rest2), optrel(Rest2,Rest).
np(Ein,Rest)        :-  pn(Ein,Rest).
optrel(Ein,Rest)    :-  Ein = Rest.
optrel(Ein,Rest)    :-  terminal(that,Ein,Rest1), vp(Rest1,Rest).
...
terminal(Wort,Ein,Rest1) := Ein = [Wort | Rest1],
%%% oder:     terminal(Wort, [Wort | Rest1], Rest1) .
```

Man erkennt, daß die Grammatik eins-zu-eins übersetzt wird, wobei nur die Behandlung der Eingabe- und Restliste hinzukommt. Diese Methode nennt man auch Verwendung von Differenzlisten, wobei man manchmal auch das Paar $(Ein, Rest)$ als Datentyp "Differenzliste" beschreibt.

Dieser Parser antwortet nur mit ja/nein bzw. terminiert nicht.

Man teilt die Grammatik normalerweise ein in die eigentliche Grammatik und das *Lexikon*, das die Terminale beschreibt. Teilweise zählt man auch die *Präterminale* wie z.B. N (Nomen) zum Lexikon.

8.12 Definite Clause Grammars

Definite Clause Grammars (DCG's) sind eine Erweiterung der kontextfreien Grammatiken um Argumente (Attribute) von Nichtterminalen. Z.B. sollte der Numerus von NP und VP übereinstimmen, d.h. beides sollte entweder im Singular oder Plural sein: "sie gehen. er geht", aber nicht "*er gehen".

Dies ergibt z.B. folgendes Grammatikfragment, in dem das Lexikon der Einfachheit halber in der DCG enthalten ist. In der Syntax der DCGs sieht das folgendermaßen aus:

```
s(Number)     -->  np(Number), vp(Number).
np(Number)    -->  pn(Number).
vp(Number)    -->  tv(Number), np(_).
```

```
vp(Number)    -->  iv(Number).
pn(singular)  -->  [shrdlu].
pn(plural)    -->  [they].
iv(singular)  -->  [halts].
iv(plural)    -->  [halt].
```

Übersetzt man in der gleichen Weise wie oben nach Prolog, so ergibt sich:

```
s(P0,P, Number)    :-  np(Number,P0,P1), vp(Number,P1,P).
np(Number,P0, P)   :- pn(Number,P0,P).
vp(Number,P0, P)   :- tv(Number, P0, P1), np(_, P1, P).
vp(Number,P0, P)   :- iv(Number, P0, P).
pn(singular,P0,P)  :- terminal(shrdlu, P0, P).
pn(plural, P0, P)  :- terminal(they,P0, P).
iv(singular, P0, P):- terminal(halts, P0, P).
iv(plural, P0, P)  :- terminal(halt, P0, P).
```

Benutzung von Differenzlisten

Verwendet man das Minuszeichen als Konstruktor für Differenzlisten, so ergibt sich:

```
s(P0 - P)          :-  np(Number,P0-P1), vp(Number,P1-P).
np(Number,P0- P)   :- pn(Number,P0-P).
vp(Number,P0- P)   :- tv(Number, P0- P1), np(_, P1- P).
vp(Number,P0-P)    :- iv(Number, P0- P).
pn(singular,P0-P)  :- terminal(shrdlu, P0-P).
pn(plural, P0- P)  :- terminal(they,P0- P).
iv(singular, P0-P) :- terminal(halts, P0-P).
iv(plural, P0-P)   :- terminal(halt, P0- P).
```

Die Definition für `terminal` ist:

```
terminal(X, [X|P] - P).
```

Die Anfrage `s([shrdlu, halts]-[])` hat folgende Verarbeitung:

```
  s([shrdlu, halts]-[])
     np(N,[shrdlu, halts]-P1), vp(N,P1-[])
     pn(N,[shrdlu, halts]-P1), iv(N,P1-[])
        terminal(shrdlu,[shrdlu, halts]-P1),  iv(singular,P1-[])
             %%%   (N = singular)
 terminal(shrdlu,[shrdlu, halts]-P1),  terminal(halts,P1-[])
             %%%  (Unifikation mit terminal ergibt P1 = [halts])
 terminal(halts,[halts]-[]).
Ergibt ''yes''.
```

Es gibt noch weitere dieser Attribute[8] z.B. Person (erste, zweite, dritte) muß übereinstimmen (ich bin; du bist; er, sie, es ist); bei NP und VP muß das Geschlecht übereinstimmen: das Haus (nicht *die Haus), ebenso gibt es eine Übereinstimmung bei Det und N.

Im Deutschen kann man aus der Endung den Casus (teilweise) ablesen mittels morphologischer Verarbeitung. Die sogenannte morphologische Verarbeitung dient dazu, Worte zu analysieren: auf Endungen, Zerlegung zusammengesetzter Worte, ... In dieser Vorlesungen werden wir darauf nicht weiter eingehen.

Bestimmte Satzkonstruktionen erfordern Dativ, bzw. Akkusativ, ... die ebenfalls als erforderliche Attribute entweder direkt in die Grammatikregeln eingefügt werden oder als entsprechende Verbmarlierungen im Lexikon Auch die Zeit oder (Aktiv/Passiv) kann man hinzufügen.

Lexikon

Aus heutiger Sicht ist das Lexikon nicht nur eine Sammlung von Worten, sondern die *zentrale Struktur* der linguistischen Verarbeitung. Man kann sehr viel bereits im Lexikon kodieren und dann mit einfachen weiteren Regeln auskommen. D.h. daß z.B.viele Eigenschaften eines Wortes im Lexikoneintrag steckt, auch solche Angaben, die besagen in welchen Satzkonstruktionen bestimmte Verben erlaubt sind. Z.B. Passiv-verbot: (*ich werde gelaufen). z.B. daß ein Verb transitiv bzw intransitiv ist oder in beiden Versionen benutzt werden kann. Die Kontextinformation, die ein bestimmtes Wort benötigt, kann im Lexikon kodiert sein. Auch inhaltliche Informationen (semantics) können dort enthalten sein.

Beispiel 8.15.

```
take: verb
 verbtype =transitive
 subject: role = agent, semfeat= human
 object: role = instrument, semfeat=vehicle
 prep-obj:  prep=to role=goal
 prep-obj: prep=from, role=source, ...
```

Berechnung von Parse-Bäumen

Man kann den Nichtterminalen und Terminalen ein Argument mitgeben, das den Parsebaum mitberechnet. (Dies geht auch in einer attributierten Grammatik.)

```
S         --->  NP VP
NP        --->  PN
Optrel    --->  []
```

[8] Attribute werden in der Computerlinguistik auch features genannt; Die zugehörigen Grammatiken Unifikationsgrammatiken

```
Optrel    --->  that VP
VP        --->  IV
```

Die Implementierung kann dafür z.B. jedes Prädikat um ein Argument erweitern, das den (berechneten) Syntaxbaum aufnimmt.

```
s(sent(TR_np, TR_vp), Ein, Rest) :- np(TR_np,...), vp(TR_vp,...).
np(nphrase(TR_pn),...) :- PN(TR_pn,...).
optrel(kein_relativesatz,...).
optrel(relativ_satz(TR_iv),...) :- iv(TR_iv,...)
```

Hier sollte z.B. der Satz "Shrdlu halts" als Ausgabe den Syntaxbaum $s(np(pn(shrdlu)), vp(iv(halts)))$ erzeugen.

DCG's als formales System

DCG's sind als formales System aufzufassen, das analog zu CFGs eine formale Sprache definiert. Zusätzlich zu CFGs sind an den Nichtterminalen Argumente erlaubt, die Variablen und Konstanten sein können und auf Gleichheit getestet werden dürfen.

Die formale Sprache die zu einer DCG gehört, entspricht genau den erfolgreichen Parses nach der Übersetzung in definite Klauseln, wenn man SLD-Resolution als operationale Semantik nimmt.

Allerdings ist das nicht dasselbe wie die Gleichsetzung mit der Prologimplementierung derselben, da nach der Übersetzung das Parsen als rekursiv absteigend festgelegt ist.

DCGs kann man zu attributierte Grammatiken dadurch abgrenzen, daß die letzteren kontextfreie Grammatiken sind, die in den Regeln Berechnungsalgorithmen für eine festgelegte Menge von Attributen enthalten, aber aufgrund der Berechnungen keine Eingabe ablehnen können.

Eine theoretische Klassifizierung zwischen DCGs und CFG bzgl der erkannten formalen Sprachen ist: Offenbar sind DCGs eine Erweiterung der CFGs. Man kann aber mit DCG's über kontextfrei hinausgehen:

DCG-Grammatik für die nicht-kontextfreie formale Sprache $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$

```
S        ---> A(x)B(x)C(x)D(x)
A(s(x))  ---> a A(x)
A(0)     ---> a
B(s(x))  ---> b B(x)
B(0)     ---> b
C(s(x))  ---> c C(x)
C(0)     ---> c
D(s(x))  ---> d D(x)
D(0)     ---> d
```

Bemerkungen zur Verarbeitung natürlicher Sprache

Eine natürliche Sprache ist i.a. so aufgebaut, daß Mehrdeutigkeiten vom Sprecher (Schreiber) vermieden werden können. D.h. daß es für einen Satz möglichst nur einen Parsebaum gibt. Bei mehr als einem Parse für einen Satz wird der gemeinte Inhalt nicht eindeutig erkannt. Z.B.

"Er sah das Mädchen mit dem Fernglas"

D.h. nicht, daß die Grammatik eindeutig sein muß; die Bildung mehrdeutige Sätze sind möglich. Allerdings werden bei der Kommunikation alle "Tricks" verwendet um Aussagen eindeutig zu machen: (Inhaltliche, Kontext der Aussage, vorhergehende Unterhaltung bzw. Text; bei gesprochener Sprache: Betonung. bei Sichtkontakt: Gesten usw.)

Es gibt Sprachen (Latein; auch Deutsch zählt dazu), Englisch eher nicht), in denen man Satzteile (Phrasen) teilweise umsortieren kann, ohne die Bedeutung des Satzes zu ändern. Dies liegt oft an den Endungen, die die Rolle der Konstituenten deutlich machen:

the man bites the dog

the dog bites the man

Der Hund beißt den Mann

den Mann beißt der Hund