Kapitel 4.
Prolog-Syntax

Dieses Kapitel soll in die etwas formaleren Grundlagen der Syntax und Semantik von Prolog einführen. Nach einem allgemeinen Teil wird am Ende des Kapitels ein für die Programmierung in Prolog sehr wichtiger Datentyp vorgestellt, nämlich die LISTE.

4.1. Syntax und Semantik formaler Sprachen

Wie bei natürlichen Sprachen unterscheidet man auch bei formalen Sprachen und Programmiersprachen zwischen der SYNTAX und der SEMANTIK von Ausdrücken (Sätzen, Texten, Programmen). Die SYNTAX einer Sprache legt durch die Regeln fest, wie die Ausdrücke der Sprache gebildet werden können. Die SEMANTIK legt die Bedeutung oder Funktion der Ausdrücke fest.

Voraussetzung für die sinnvolle Verarbeitung eines Programmes ist seine syntaktische Korrektheit. Falls ein Ausdruck gegen eine Regel oder mehrere Regeln verstößt, kann er nicht weiterverarbeitet werden, d.h. ihm kann keine Bedeutung oder Funktion zugeordnet werden. Dies gilt für natürliche Sprachen wie Englisch oder Deutsch genauso wie für formale, z.B. für die Prädikatenlogik erster Stufe oder für Prolog. Wenn also ein Programm nicht das tut, was es tun sollte, bietet es sich an, zunächst die Syntax zu prüfen.

Ein Programm ist SEMANTISCH korrekt, wenn es das tut, was es tun soll. Die syntaktische Korrektheit ist eine notwendige, jedoch keine hinreichende Bedingung für die semantische Korrektheit eines Programmes. Mit anderen Worten, ein syntaktisch korrektes Programm kann semantisch abweichend sein.

4.2. Die Syntax von Prolog: Das Alphabet

Jede Sprache verfügt über einen Vorrat von Grundzeichen, aus denen die komplexen Ausdrücke nach festen Regeln aufgebaut sind. Dieser elementare Zeichenvorrat wird ALPHABET genannt.[18] Für diesen Kurs relevant ist dabei besonders die Tatsache, daß weder Umlaute noch das 'ß' zum Prolog–Alphabet zählen. Umlaute müssen als Kombination von Vokal + 'e', das 'ß' entsprechend als 'ss' ausgedrückt werden.

Das gesamte Alphabet der Grundzeichen ist in ZEICHENKLASSEN eingeteilt, die für die Syntaxregeln relevant sind:

- Kleinbuchstaben: a .. z
- Großbuchstaben: A .. Z
- Ziffern: 0 .. 9
- Sonderzeichen: !, #, $, %, &, *, +, ,(Komma), –, . , /, :, ;, <, >, ?, @, |, \, ~, {, }
- Unterstreichungszeichen: _
- Trennzeichen: (Leerstelle), ", ', (,), , (Komma), [,]
- Steuerzeichen: ASCII 0 bis ASCII 31, ASCII 127[19]

Ein Prologprogramm, das in Textform vorliegt, muß vor der Verarbeitung durch Prolog zuerst gelesen und in eine interne Repräsentationsform übersetzt werden. Der Text wird zeichenweise gelesen — dies geschieht durch ein Programm, das *Scanner* genannt wird — und bestimmte regelhaft gebildete Zeichenfolgen werden als "lexikalische" Einheiten erkannt. Es können zunächst zwei Hauptklassen voneinander unterschieden werden:

---

[18] Dem Alphabet von Prolog liegt der ASCII-Zeichensatz zugrunde, und zwar standardmäßig im ursprünglichen 7-Bit-Code (ASCII 0 .. 127). *ASCII* ist ein Akronym aus den Anfangsbuchstaben des Ausdrucks ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange.* In diesem Code besteht beispielsweise eine systematische Beziehung zwischen Groß- und Kleinbuchstaben: ASCII(Klein) = ASCII(Groß)+32. Der Buchstabe *A* hat den ASCII-Code 65 und *a* den ASCII-Code 97.

[19] Sie heißen Steuerzeichen, weil sie sie nicht primär zur Textdarstellung verwendet werden (sie sind nicht druckbar), sondern um bestimmte Aktionen zu bewirken.

SYMBOLE sind Zeichenfolgen, die mit einem Buchstaben beginnen und nur Buchstaben, Ziffern, oder das Unterstreichungszeichen enthalten, z.B. Anton, maennlich, B22, verheiratet, a_A, ach_so; oder aber Folgen von Sonderzeichen wie z.B. >==<; /%&@

ZAHLEN sind entweder Ganzzahlen wie 123, die nur aus Ziffern bestehen, oder Dezimalzahlen mit Dezimalpunkt (12.5), oder Dezimalzahlen aus Mantisse und Exponent (1.35e23, 3.14e-3).

Die ersten Fehler können schon beim zeichenweisen Lesen auftreten, nämlich dann, wenn im Text Zeichen enthalten sind, die nicht zum zugelassenen Alphabet gehören. In SWI-Prolog werden über die Tastatur eingegebene Zeichen, die nicht zugelassen sind, nicht akzeptiert.

4.3. Die Syntax von Prolog: Terme

Komplexe syntaktische Ausdrücke sind aus den Elementen des Alphabets nach festen Regeln aufgebaut. Das Programm, das beim Einlesen die syntaktische Analyse vornimmt, wird *Parser* genannt. Syntaktisch nicht korrekte Ausdrücke werden als Syntaxfehler (*Syntax error*) gemeldet.

Logikprogramme bestehen aus TERMEN. TERME sind somit die syntaktischen Grundbausteine von Prologprogrammen. Ein wichtiger Punkt dabei ist, daß Terme entweder einfach oder komplex sein können. Komplex heißt nicht mehr, als daß diese Terme selber aus anderen Termen zusammengesetzt sind. Einfache Terme lassen sich zunächst wie folgt untergliedern; wie wir sehen, haben wir einige der nachfolgende Elemente bereits im zweiten Kapitel kennengelernt:

Einfache Terme
einfache Terme unterteilen sich in Konstante einerseits und Variable andererseits.

Konstante
unterteilen sich in Atome einerseits und Zahlen andererseits.

Atome
sind Symbole, die entweder mit einem Kleinbuchstaben beginnen (z.B. kasimir, tisch25, nomen, ist_ein) oder aus einer in Hochkommata eingeschlossenen beliebigen Zeichenfolgen (z.B. '123abc', 'Anton', 'Peter der Grosse') oder aber Folgen von Sonderzeichen.

Zahlen
sind entweder Ganzahlen (*integer*, z.B. 25, 123, 5000) oder Dezimalzahlen (mit Dezimalpunkt: 3.14 oder in Exponentenschreibweise: 1.34e10).

Variable
Variable untergliedern sich in anonyme und nicht-anonyme Variable

nicht–anonyme Variable
sind Zeichenfolgen, die mit einem Großbuchstaben beginnen, z.B. XYZ, Kind usw. Es kann ein einziger Buchstabe sein oder eine Buchstaben-Zahlen Kombination(z.B. X, E1, Person2).

anonyme Variable:
sind entweder Zeichenfolgen, die mit einem Unterstrich beginnen (z.B. _xyz, _weiblich usw.) oder nur der Unterstrich '_'.

Die einfachen Terme bilden zunächst das Grundmaterial, um komplexere Terme zu bilden. Ein einfaches Beispiel für einen solchen komplexen Terme ist z.B. der Prolog-Ausdruck maennlich(Person), den wir bereits aus dem vorigen Kapitel kennen. Hier wurde ein Atom (maennlich) mit einer nicht-anonymen Variablen (Person) in einer Funktor–Argument–Struktur zu einem komplexen Term kombiniert.

Komplexe Terme

Komplexe Terme werden in Form von Funktor–Argument–Strukturen notiert und entsprechend STRUKTUREN genannt. Sie bestehen aus einem FUNKTOR und einer beliebigen Anzahl von ARGUMENTEN, die ihrerseits wieder Terme sind. Beispiel:

```
satz   (nominalphrase  , verbalphrase)
  ↑          ↑                ↑
Funktor    Argument         Argument
```

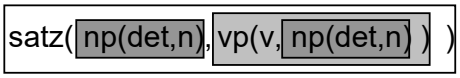
Hier ist satz ein Funktor mit zwei Argumenten, nämlich nominalphrase und verbalphrase, die ihrerseits Atome (Konstanten) sind. Es ist erkennbar, daß die Prädikat-Argument-Strukturen von Fakten Strukturen in diesem allgemeinen Sinne sind.

KONSTANTE werden zur Bezeichnung spezifischer Objekte oder Relationen verwendet. In Standardprolog gibt es von Haus aus nur zwei Arten von Konstanten: ATOME und ZAHLEN (ursprünglich sogar nur Ganzzahlen *integer*).[20]

Da die Argumente eines zusammengesetzten Terms ihrerseits Terme sind, die selbst einfach oder komplex sein können, haben wir es mit REKURSIVEN Strukturen zu tun. Strukturen sind rekursiv, wenn ihre Bestandteile nach dem gleichen Muster aufgebaut sind wie das Ganze. Betrachten wir dazu ein Beispiel:

satz(np(det,n), vp(v, np(det,n)))

Diese Struktur besteht zunächst aus dem zweistelligen Funktor satz und zwei Argumenten, np(det,n) und vp(v,np(det,n)), die ihrerseits komplexe Terme sind. Der äußerste Funktor einer Struktur wird HAUPTFUNKTOR genannt. In unserem Beispiel ist also satz der Hauptfunktor. Durch die folgende Darstellung dieses Ausdruckes wird der Aufbau deutlich:



Der Begriff TERM kann daher am besten REKURSIV bzw. INDUKTIV definiert werden. Eine Definition ist rekursiv, wenn der zu definierende Begriff in der Definition selbst verwendet wird, ohne dadurch zirkulär zu werden.

Definition 4.1. *Term*

1. Variable sind Terme
2. Konstante sind Terme
3. Ist f ein n-stelliger Funktor und sind $t_1, \ldots, t_n$ Terme, dann ist $f(t_1,\ldots,t_n)$ ein Term. (Rekursionsschritt).
4. Nur nach (1) – (3) gebildete Ausdrücke sind Terme.

In einem Beispiel wollen wir betrachten, wie nach dieser Definition komplexe Terme gebildet werden können, indem wir den Term np(det(the),noun(boy)) aus den oa. Regeln ableiten:

Zeile	Term	Ableitung
(1)	boy	R(egel) 1
(2)	the	R 2
(3)	noun(boy)	(2), R 3
(4)	det(the)	(3), R 3
(5)	np(det(the),noun(boy))	(5),(4), R 3

STRUKTUREN sind komplexe Terme und können auf verschiedene Weise verwendet werden. Als Argumente sind sie normalerweise uninterpretierte Ausdrücke und können daher zur symbolischen Darstellung beliebiger Objekte verwendet werden. Vergleiche:

1. Ophelia ist eine Katze → katze(ophelia).
2. Bart füttert die Katze Ophelia → füttert(bart, katze(ophelia))

[20] In neueren Implementierungen von Prolog gibt es darüber hinaus noch weitere einfache Datentypen, z.B. Konstante für Zeichen und Zeichenketten. Für die Eingabe von der Tastatur gibt es eine besondere Notation für Zeichen und Zeichenketten:
Zeichen: 0'a, 0'b, ..., 0'z
Zeichenketten: "Das ist toll", "Hans liebt Maria"
Diese werden jedoch intern sofort in Zahlen bzw. Zahlenfolgen umgewandelt. Die Schreibweise "abc" ist nur eine besondere Notation für [97, 98, 99].

Im ersten Fall wird die Struktur katze(ophelia) zur Darstellung des Fakts *Ophelia ist eine Katze* verwendet, im zweiten Fall zur Darstellung des Individuums *die Katze Ophelia.* Bei komplexen Strukturen wird der äußerste Funktor (in Beispiel 2: *füttert*) als HAUPTFUNKTOR bezeichnet.

In den nächsten Abschnitten dieses Kapitel werden einige eingebaute Prädikate vorgestellt, die dazu dienen, Terme einerseits zu überprüfen und, was wesentlich ist, Terme zu manipulieren. Bevor wir uns diese Prädikate näher ansehen, sollen allerdings die Notationsgrundlagen vorgestellt werden, mit denen wir im weiteren Verlauf des Skriptes arbeiten.

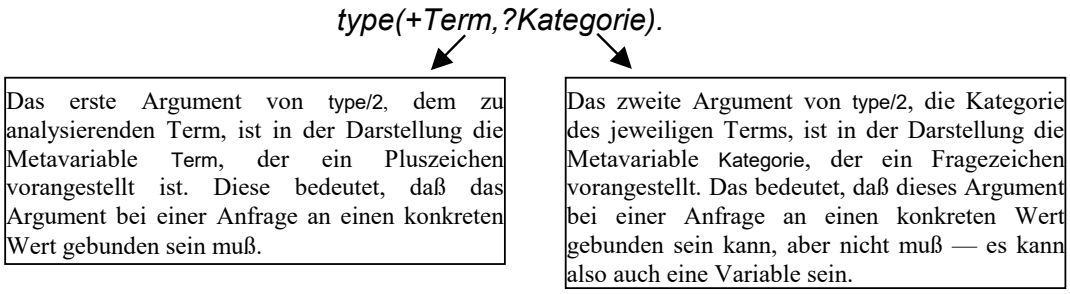
4.3.1. EXKURS: NOTATIONSKONVENTIONEN FÜR PRÄDIKATE

Als Beispiel dient das eingebaute Prädikat type/2, welches überprüft, zu welcher Kategorie ein beliebiger Term angehört. Das erste Argument von type/2 ist der Term, den es zu überprüfen gilt, das zweite Argument ist der Name der Kategorie, der der Term angehört. Beispiel: eine Anfrage wie type(hallo,atom).

wird die Ausgabe Yes erhalten und also bestätigt. Natürlich kann das zweite Argument auch eine Variable sein, wie beispielsweise in type(3,X). Darauf erscheint die folgende Antwort:

X = integer.

Das Prädikat type/2 wird also mit einem beliebigen Term als erstem Argument und als zweitem Argument entweder einer konkreten Angabe (atom, integer, string usw.) oder einer Variablen aufgerufen. Um diesen Sachverhalt kurz und bündig zu erfassen, werden Metavariablen und die Zeichen '+' und '?' verwendet:



Wenn im fortlaufenden Skript Prädikate vorgestellt werden, und den Argumenten des Prädikates (in Form von Metavariablen) die Zeichen '+', '?' und '-' vorangestellt werden, haben diese für die konkrete Verwendung des Prädikates jeweils die folgende Bedeutung:

+Argument Das Argument muß an einen konkreten Wert gebunden sein (es darf keine Variable sein)

–Argument Das Argument darf nicht an einen konkreten Wert gebunden sein (es muß eine Variable sein)

?Argument Das Argument kann, muß aber nicht an einen konkreten Wert gebunden sein (es kann eine Variable sein, muß aber nicht)

Eine einfachere Darstellung dafür sieht schlicht so aus:

type(+,?).

4.3.2. PRÄDIKATE ZUR TERMMANIPULATION

Wir werden später in vielen Beispielen sehen, daß es besonders für linguistische Fragestellungen wichtig ist, die Struktur von Termen in verschiedener Weise analysieren und manipulieren zu können. So werden wir Strukturen zur Darstellung von Strukturbeschreibungen in der Syntax verwenden. Für einen ersten Einstieg wollen wir zunächst zwei wichtige Systemprädikate vorstellen, die zur Manipulation von Termen dienen: die Prädikate functor/3 und arg/3 (vgl. Handbuch).[21]

[21] Das Handbuch ist noch in Arbeit, weswegen hier auch Seitenangaben verzichtet werden muß.

functor/3

Das Systemprädikat functor/3 stellt eine Beziehung her zwischen einem Term (1. Argument), seinem Funktor (2. Argument) und seiner Stelligkeit (3. Argument):

functor(*?Term, ?Funktor, ?Stelligkeit*).

Es gilt z.B. die Beziehung

functor(f(a,b), f, 2).

Die Anfrage

?- functor(g(2, 3),F, N),

wobei F und N Variable sind, liefert als Lösung F=g, N=2.

Es sind einige Sonderfälle zu beachten. Was geschieht z.B., wenn man als erstes Argument keine Struktur angibt, sondern ein Atom? In diesem Fall wird das Atom als eine 0-stellige Struktur interpretiert, mit dem Atom als Funktor und 0 Argumenten:

```
?- functor(nomen,F,Stelligkeit).
F = nomen
Stelligkeit = 0
yes
```

Interessant ist der Fall, wo das erste Argument, der Term, eine Variable ist. Die beiden anderen Argumente müssen dann gebunden sein. Ist die Stelligkeit 0, erhalten wir folgendes:

```
?- functor(S, verb, 0).
S=verb
yes
```

Ist die Anzahl der Argumente > 0, dann wird ein Strukturmuster mit gegebenem Funktor und einer der gegebenen Stelligkeit entsprechenden Anzahl von Variablen als Argumenten erzeugt. Beispiel:

```
?- functor(Term, np,3)
Term=np(_G33,_G34,_G35)
yes
```

In Verbindung mit dem folgenden Systemprädikat lassen sich mit Komponenten aus unterschiedlichen Quellen neue Strukturen aufbauen.

arg/3

Das Systemprädikat arg/3 (Syntax: arg(*?N, ?Term, ?Argument*) ermittelt das n-te Argument eines gegebenen Terms. Es gilt z.B.

```
?- arg(2, np(det, n), n)
yes

?- arg(2, vp(vt, np(det, n), Arg)
Arg = np(det, n)

yes
```

Ist das n-te Argument von *Term* eine Variable, so wird sie gegebenenfalls an den Wert von *Argument* gebunden, z.B.

```
?- arg(1, satz(NP,vp(v np)), np(det, n))
NP = np(det, n)
yes
```

4.3.3. TERMKLASSEN

Es können verschiedene Klassen von Termen unterschieden werden. Da Variable während der Berechnung an beliebige Objekte gebunden werden können, ist es sinnvoll, Prädikate zur Verfügung zu haben, die die Zugehörigkeit eines Objekts zu einer Termklasse überprüfen.

Wie schon erwähnt, können wir zunächst zwischen den Klassen KONSTANTE, VARIABLE und STRUKTUREN unterscheiden. Strukturen stehen zwischen den Konstanten und Variablen, weil sie Variable

enthalten können. In SWI-Prolog können Konstante in die Unterklassen ATOME, GANZZAHLEN, und DEZIMALZAHLEN unterteilt werden:

TERMKLASSEN

- Konstanten
  - Atome: boy, 'the boy', $%,,%

  Zahlen:

  Ganzzahlen: 123, 589

  Dezimalzahlen: 1.75 bzw. 2.05
- Variablen: X, Y, Abc,_blau, _
- Strukturen: np(det,n),wort(form(singt),stamm(sing),morph(p3,sg))
- Listen: [hans, ist, np], [[der, mann], [ist, [im, [garten]]]].

Im folgenden stehen die syntaktischen Eigenschaften von Prolog-Termen im Vordergrund. Aber zusätzlich wird darauf eingegangen, welche semantischen Rollen die einzelnen Termklassen vorrangig spielen. Mit semantischer Rolle ist die Funktion im größeren Gesamtzusammenhang oder auch die mögliche Bedeutung eines Prolog-Terms gemeint.

4.3.4. ATOME

1. Regel für Atome:

Zeichenfolgen, die mit einem Kleinbuchstaben beginnen und nur Klein- und Großbuchstaben, Zahlen und das Unterstreichungszeichen enthalten, sind Atome.

Beispiele:

satz, np, vp etc. , ein_ziemlich_langes_Atom, boy1, jump54, null8_15

Der Unterstreichungsstrich wird vor allem verwendet, um zur besseren Lesbarkeit in längeren Atomnamen die Wortgrenzen zu markieren.

2. Regel für Atome

Jede beliebige Zeichenkette, die zwischen zwei Hochkommata (') eingeschlossen ist, behandelt das System als Atom.

Buchstaben, Zahlen und Sonderzeichen können zwischen Hochkommata beliebig gemischt werden. Die Hochkommata gehören selbst nicht zum Atom, sondern begrenzen es nur. Innerhalb von Hochkommata können fast ausnahmslos sämtliche Zeichen von 0 bis 127 vorkommen.

Die einzige Ausnahme bildet das Hochkomma selbst, das innerhalb von Atomen in Ersatzdarstellung angegeben werden muß, und zwar durch Doppelschreibung: z.B. 'John''s'.

Beispiele:

'Satz', '123 los'

'Aller Anfang ist schwer'

'John''s house'

Manchmal ist es zweckdienlich, Zeichenreihen, die Prolog als Atome verstehen soll, im Zweifelsfall mit Hochkommata einzugeben. Allgemein läßt sich sagen: Wenn ein Ausdruck A syntaktisch ein Atom ist, so ist auch 'A' ein Atom, das mit A identisch ist. Beispiel:

```
?- atom = 'atom'.
yes
```

Hochkommata können also nie schaden.

3. Regel für Atome

Sämtliche Zeichenketten aus einer beliebigen Anzahl von Sonderzeichen sind Atome.

Beispiele:

```
>==<
?/%&@
```

So gebildete Atome werden meist zur Bezeichnung von OPERATOREN verwendet.[22]

Testprädikate atom/1 *und* atomic/1

Zur Überprüfung, ob ein Term ein Atom ist, dient das Systemprädikat atom/1. Es akzeptiert einen beliebigen, korrekten Prolog Term als Argument und prüft, ob es sich um ein Atom handelt oder nicht. Falls Sie prüfen wollen, ob () ein Atom ist, gibt Prolog allerdings statt *No* nur eine Syntaxfehler-Meldung aus, da () kein korrekter Term ist. Einige Beispiele:

```
?- atom(variable), atom ('???').
yes
?- atom(123).
No
```

Etwas anders verhält sich das Prädikat atomic/1, das sowohl bei einem Atom als auch bei einer Zahl (Ganzzahl oder Dezimalzahl) yes zur Antwort gibt.

Beispiel:

```
?- atomic(xxx), atomic(222).
yes
```

Semantische Rollen von Atomen

Die möglichen semantischen Rollen, die Atome spielen können, sind recht verschieden. Atome dienen für gewöhnlich dazu, bestimmte abstrakte oder konkrete Objekte aus dem Problemkreis, den ein Prolog Programm beschreibt, zu bezeichnen (wie z.B. hans, auto, nominalphrase, satz). Atome und nur Atome dienen als Bezeichner für Funktoren wie z.B. nomen in nomen(kaffee). Eine gewisse syntaktische Ambiguität entsteht aus der Tatsache, daß auch nullstellige Funktoren zugelassen sind. Der Sachverhalt, daß es regnet, könnte so als Faktum dargestellt werden: es_regnet. Im Deutschen gibt es sogenannte Witterungsprädikate. In so einem Fall gibt es kein sinnvoll bestimmbares Objekt, dem durch die Aussage, daß es regnet, daß es schneit usw. eine Eigenschaft zugewiesen würde. Somit ist die Prolog-Formalisierung es_regnet (gleichsam eine nullstellige Relation) gefühlsmäßig sicher einleuchtender als

```
regnet(es).
regnet(himmel).
regnet(luft).
```

oder ähnliches.

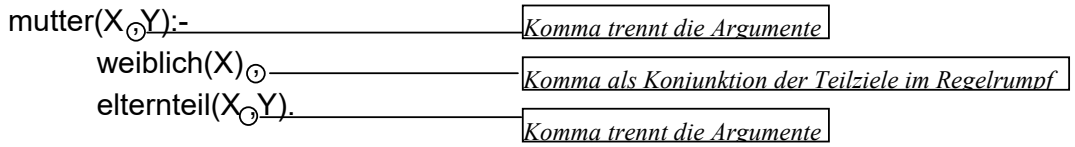
Eine besondere Sorte von Atomen heißt OPERATOREN, die nichts weiter als syntaktisch abweichend auftretende Struktur-Funktoren sind.

Probleme mit Atomen aus Sonderzeichen

Auch beim Zeichen ',' (Komma) liegt eine Ambiguität vor. Es fungiert einmal als normales Sonderzeichen und ist als Operator zur Konjunktion (*=und-Verknüpfung*) von Zielklauseln definiert. Zum anderen dient das Komma ',' als Separator von Argumenten innerhalb von Strukturen (siehe unten). Dies führt dazu, daß in bestimmten Kontexten die Rolle von ',' zweideutig ist.

Beispiel: in der folgenden Regel haben die Kommata unterschiedliche Funktionen:

[22] Näheres dazu weiter unten.



Ebenso ist mit dem Atom '.' (Punkt) Vorsicht geboten, denn der Punkt dient auch dazu, in einem Programm aufeinanderfolgende Klauseln voneinander zu trennen.

4.3.5. ZAHLEN

Die Darstellung von Zahlen in Prolog weicht kaum von anderen Programmiersprachen ab. Es gibt ganze Zahlen (*integers*) und Dezimalzahlen (reelle Zahlen, *reals*) mit oder ohne Vorzeichen. Es gibt eine ganz Reihe von Testprädikaten für Zahlen (integer/1, float/1, number/1, atomic/1), siehe dazu aber das Handbuch.

4.3.6. VARIABLE

Regel für Variablen

Jede Zeichenkette, die mit einem Großbuchstaben A .. Z oder mit dem Unterstrich _ beginnt und nur Klein- und Großbuchstaben sowie Ziffern enthält, ist eine Variable.

Beispiele:

ATOM, X, Y

_atom, _x, _y

Testprädikate var/1 und nonvar/1

Jedes Prolog-System stellt zwei Testprädikate zur Verfügung, mit denen sich prüfen läßt, ob ein Term semantisch eine Variable ist oder nicht. Die Testprädikate lauten var/1 (testet, ob ein Ausdruck eine Variable ist) und nonvar/1 (testet, ob ein Ausdruck keine Variable ist). Beispiele:

```
?-  var(X).
    X = _G105
    Yes
?-  nonvar(X).
    No
?-  nonvar(abc).
    Yes
?-  var(X),        /* X ist hier noch ungebunden*/
    X = atom,     /* X wird an den Wert 'atom' gebunden*/
    nonvar(X).    /* X ist jetzt gebunden*/
    X = atom
    yes
```

Erläuterung zur letzten Anfrage: beim Aufruf von var/1 handelt es sich bei X um eine Variable, da X syntaktisch wie eine Variable strukturiert ist und nicht an einen nichtvariablen Term gebunden ist. Durch den Aufruf von =/2 wird X an das Atom atom gebunden. Der Aufruf von nonvar/1 glückt, weil er an dieser Stelle gleichbedeutend ist mit dem Aufruf von nonvar(atom). Damit ein Term von den Testprädikaten als Variable erkannt werden kann, muß er wirklich eine ungebundene Variable sein.

Semantische Rolle von Variablen

Variablen erfüllen die Rolle von Platzhaltern. Variablen können jeden beliebigen Prolog-Ausdruck repräsentieren, indem sie an ihn gebunden werden.

Eine Sonderrolle spielt die anonyme Variable '_'. Sie besteht aus einem einzigen Unterstrich. Der einfache Unterstrich ist unter allen Umständen eine Variable und kann nicht dauerhaft an einen bestimmten Wert

gebunden werden. Gleichnamige Variablen repräsentieren in Prolog innerhalb einer Klausel stets denselben Wert. Dies gilt nicht für die Variable '_'. Daher die Bezeichnung “anonyme Variable”. Sie kann auch für verschiedene Objekte innerhalb einer Klausel oder Anfrage stehen.

Variable, die aus Zeichenfolngen bestehen, die mit einem Unterstrich beginnen, sind ebenfalls anonym insofern, als sie zwar gebunden werden können, diese Bindung aber nicht als Lösung ausgegeben wird. Vergleiche dazu das unterschiedliche Verhalten in den folgenden Beispielen:

```
?- X = bart, _ = lisa, Y=_.
X = bart
Y = _G230
yes
?- X = bart, _X=lisa, Y=_X.
X = bart
Y = lisa
yes
```

4.3.7. STRUKTUREN

Der allgemeinste und ausdruckskräftigste Datentyp von Prolog ist die Struktur. Zwei Typen von Strukturen sind zu unterscheiden:

1. Funktor-Argument-Strukturen
2. Operator-Operand-Strukturen

Funktor-Argument-Strukturen

Funktor-Argument-Strukturen bestehen aus einem Funktor und aus einer anschließenden Klammer, die eine Folge von Argumenten enthält. Der Funktor muß ein Atom sein, die Argumente können beliebige Prolog-Terme sein. Schematisch dargestellt:

$Funktor(Argument_1,...,Argument_n)$

Beispiele:

```
verb(singt)
menu(Vorspeise, Hauptspeise, Dessert)
baum(baum(Links),knoten,baum(Rechts))
is(10, +(2, 8))
```

Wie aus den letzten Beispielen ersichtlich ist, können die Argumente von Strukturen selbst wieder Strukturen sein. Wie oben gezeigt handelt es sich bei dem Datentyp Struktur um einen rekursiven Datentyp. Wenn ein echter Teil eines Terms vom selben Typ ist wie der gesamte Term, spricht man von Rekursivität. Es gibt allenfalls hardware-spezifische Grenzen für die Anzahl der Argumente. In diesem Sinn sind Strukturen der allgemeinste und ausdrucksstärkste Datentyp von Prolog.

Operator-Operand-Strukturen

Da nun bestimmte Funktor-Argument-Strukturen — wie z.B. arithmetische Ausdrücke — schwer lesbar sind, gibt es OPERATOR-OPERAND-STRUKTUREN, die eine rein syntaktische Variante von Funktor-Argument-Strukturen darstellen, ohne daß das Prolog-System intern eine semantische Unterscheidung treffen würde. Intern werden Strukturen immer als Funktor-Argument-Strukturen repräsentiert. Beispiele sind etwa arithmetische Ausdrücke, z.B. a + b:

```
   a          +          b
   ↑          ↑          ↑
<Operand>  <Operator>  <Operand>
```

Der Unterschied zwischen Funktor-Argument-Struktur und Operator-Operand-Struktur besteht allein darin, daß die Operator-Struktur aufgrund ihrer äußeren Erscheinungsform häufig leichter lesbar ist. Dabei entsprechen sich jeweils einerseits Funktor und Operator, andererseits Argumente und Operanden.

Beispielsweise sind die Ausdrücke 2 + 3 und +(2, 3) äquivalent, sie unterscheiden sich nur in ihrem äußeren Erscheinungsbild.

Die Ausgabeprädikate write und display unterscheiden sich in der Behandlung von Strukturen. Ist ein Funktor als Operator definiert, so berücksichtigt write die syntaktischen Eigenschaften des Operators, währen display Strukturen grundsätzlich in der Funktor-Argument-Schreibweise ausgibt:

```
?- write(a + b), nl.
a + b
yes
?- display(a + b), nl.
+(a, b)
yes
```

Beispiel:

```
?- is(X,+(–(*(23,8),10),3)).
X = 177
yes
```

kann dargestellt werden als

```
?- X is 23 * 8 – 10 + 3.
X = 177.
yes
```

was ohne Zweifel erheblich übersichtlicher ist. Und wie zu sehen ist, liefert Prolog für beide Ausdrücke dasselbe Ergebnis. Es lassen sich folgende allgemeine Bauschemata angeben, wobei alle Operanden Terme sind:

1. Struktur mit Präfix-Operator:
   <Operator> <Operand> z.B. not Term
2. Struktur mit Postfix-Operator:
   <Operand> <Operator>
3. Struktur mit Infix-Operator: z.B. 2 + 3
   <Operand> <Operator> <Operand>

Die Atome is, +, – und * sind allesamt Infix-Operatoren. Durch sie können zweistellige Strukturen gebildet werden, indem sie zwischen die Argumente der Struktur geschrieben werden. Außer Infix-Operatoren gibt es noch Präfix- und Postfix-Operatoren, mit deren Hilfe einstellige Strukturen gebildet werden können. Die Operatoren – und + fungieren nicht nur als Infix- sondern auch als Präfix-Operatoren. Sie haben im Zusammenhang mit Zahlen Vorzeichenfunktion. Mit einem Postfix-Operator wird eine einstellige Struktur gebildet, indem er hinter einen Term geschrieben wird. Im übrigen sind auch Regeln zweistellige Operator-Operand-Strukturen.

```
unterbegriff(X,Y) :–
        ist_ein(X,Y).
```

ist dasselbe wie

```
:– (unterbegriff(X,Y),ist_ein(X,Y)).
```

Letzteres ist nur erheblich schlechter lesbar. In Prolog existiert die Möglichkeit, Operatoren selbst zu definieren. Auf diese Möglichkeit wird in einem späteren Kapitel eingegangen.

Semantische Rollen von Strukturen

Wie der Name besagt, dienen Strukturen dazu, strukturierte Objekte darzustellen. Strukturierte Objekte, z.B.

```
haus(zimmer1(stuhl), zimmer2(3_stuehle))
```

werden formal Relationen genannt, zu denen auch Eigenschaften wie weiblich(eva) gezählt werden. Eine typische Relation ist: sehen(X,Y) Damit kann eine zweistellige Beziehung zwischen einem X und einem Y repräsentiert sein, wobei X Y sieht. Strukturen werden insbesondere dazu verwendet, um Klauseln zu

formulieren. Die Tatsache, daß Peter morgen kommt, ließe sich als Faktum so formulieren: morgen(kommen(peter)). Der Sachverhalt (hier ein Ereignis), daß Peter kommt, wird zum Objekt gemacht, auf das der Funktor morgen wie eine Eigenschaft angewendet wird. Ob eine solche Darstellung allerdings ratsam ist, hängt natürlich von der Aufgabenstellung ab, und in einem anderen Zusammenhang mag die Darstellung desselben Faktums vielleicht so lauten: kommen(peter,morgen). In dieser Darstellung ist der Sachverhalt so interpretiert, daß es sich um eine Relation von Peter und einer Zeitangabe (morgen) handelt.

4.4. Rekursive Strukturen: Listen

Zum Abschluß dieses Kapitels soll ein Datentyp eingeführt werden, nämlich die Liste, die eigentlich auch zu den Strukturen zu rechnen ist, sich aber dadurch auszeichnet, daß es sich hier um eine rekursive Struktur handelt. Da dieser Datentyp für die Prolog-Programmierung absolut essentiell ist, ist ihm ein eigener Abschnitt gewidmet.

Der Begriff 'Rekursion' ist ja bereits eingeführt worden – zum einen im Zusammenhang mit rekursiven Prädikaten (unterbegriff/2, vorfahr/2, roemer/2), zum anderen im Zusammenhang mit der Definition von *Term* (weiter oben in diesem Kapitel).

Rekursive Strukturen nun sind Strukturen, deren Komponenten nach dem gleichen Muster gebaut sind wie das Ganze. Damit solche Strukturen endlich sind, müssen sie auch für einfache Fälle definiert sein. Die Möglichkeit von rekursiven Strukturen in Prolog ist (wie gesehen) in der Termdefinition angelegt. Die wichtigste rekursive Struktur in Prolog ist die LISTE. Sie ist ein vordefinierter Datentyp, für den es eine besondere Notationsweise gibt. Informell ist eine Liste eine Folge von beliebigen Termen, die selbst Listen sein können. Die Terme werden durch Kommata voneinander getrennt und in eckige Klammern eingeschlossen: [1, bart, "Lisa", np(det, n), [a, b]]. Diese einfache Notation darf aber nicht darüber hinwegtäuschen, daß Listen rekursive Strukturen sind, die wie folgt definiert sind:

Definition 4.1. *Liste*

Die leere Liste [ ] ist eine Liste.

Ist *K* ein Term und *R* eine Liste, dann ist die Struktur .(*K,R*) eine Liste. Für (*K,R*) schreibt man [*K* | *R*]. Man nennt *K* den KOPF und *R* den REST der Liste. | ist der Listenoperator, der Kopf und Rest einer Liste voneinander trennt.

Nur nach (1.) und (2.) gebildete Strukturen sind Listen.

Die Darstellung .(*K, R*) mit dem Punkt als Listenfunktor ist die interne Repräsentation von Listen in Standardprolog.

Beispiel für die Konstruktion einer Liste:

Zeile		aus Zeile	durch Regel
(1)	[]		R 1
(2)	[lisa | [ ]]	(1)	R 2
(3)	[liebt | [lisa | [ ]]	(2)	R 2
(4)	[bart | [liebt | [lisa | [ ]]]	(3)	R 2

Die Schreibweise [bart | [liebt | [lisa | [ ]]] ist aber sehr unübersichtlich. Deshalb gibt es eine Konvention zur Vereinfachung:

X | [Y] → X,Y

X | [ ] → X

Damit gilt [bart | [liebt | [lisa | [ ]]]] → [bart, liebt, lisa]

Listen sind also Baumstrukturen mit binären Verzweigungen:

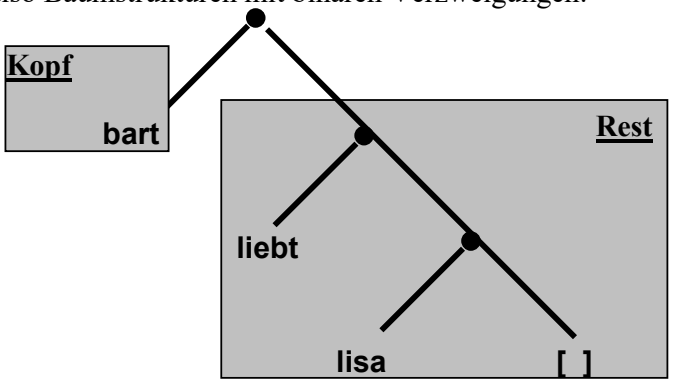


Abb. 4.2.

Eine Liste ist demnach aus lokalen Binärbäumen aufgebaut, an deren linkem Zweig ein beliebiger Prologterm als KOPF "hängt", während der rechte Zweig immer eine Liste sein muß. Diese Liste ist entweder die leere Liste als Blatt, dann gibt es keine weiteren Verweigungen, oder ein weiterer lokaler Binärbaum, bestehend aus einem Term und einer weiteren Liste, etc. Insofern kann die obenstehende Grafik wie folgt präzisiert werden:

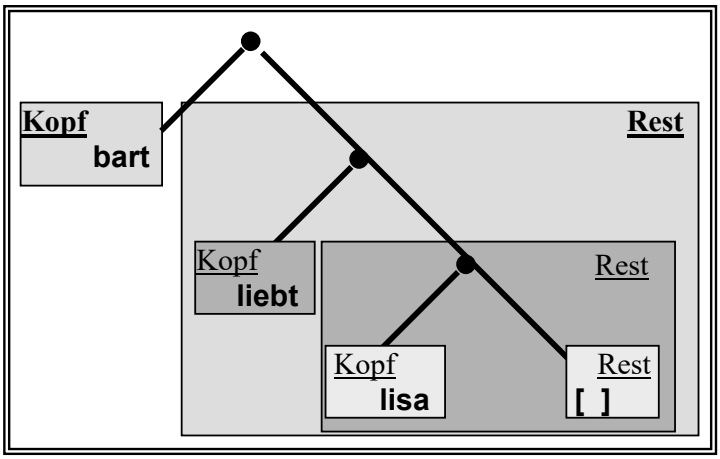


Abb. 4.3.

Ein weiteres Beispiele:

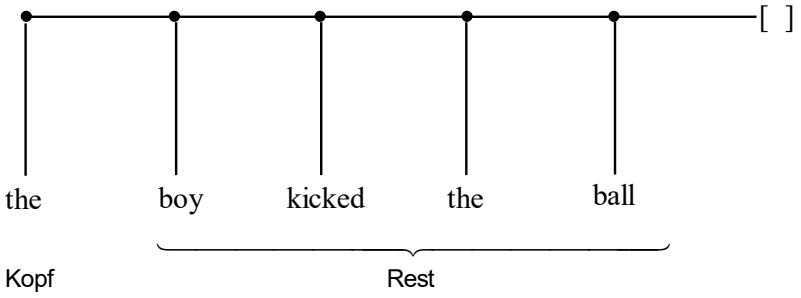


Abb. 4.4. Binnenstruktur von [the, boy, kicked, the, ball]

Listen sind also binäre (zweistellige) Strukturen. Als Struktur gedacht, haben Listen immer zwei Argumente, einen Kopf und einen Listenrest. Während der Kopf ein beliebiger Prologterm sein kann, muß das zweite Argument immer eine Liste sein, die gegebenenfalls leer sein kann. Da der Listenkopf ein beliebiger Term ist, kann er auch eine Liste sein. Der Ausdruck [[the, boy], [loves, mary]] ist ebenfalls eine Liste. Sie besteht aus zwei (!) Elementen, dem Kopf [the, boy], der selbst eine Liste ist, und dem Element [loves, mary]. Um sich über den Status dieses Elements in der Liste klar zu werden, muß man

sich die Beziehung [K, R] = [K|[R]] vor Augen halten. Mit anderen Worten, [loves, mary] ist als ganzes Element einer Liste:
[[the, boy], [loves, mary]] = [[the, boy] | [[loves, mary]]] = [[the | [boy]] | [[loves | [mary]]]].
Auch eine einelementige Liste besteht aus zwei Teilen, und zwar aus dem einen Element als Kopf und der leeren Liste als Rest: [boy] = [boy | [ ]].

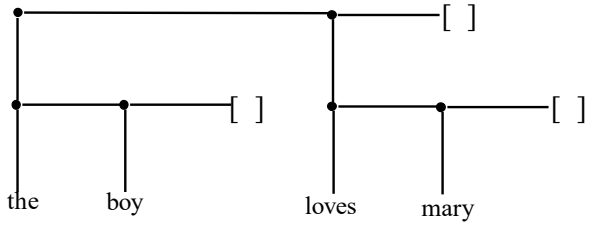


Abb. 4.5. [[the, boy], [loves, mary]]

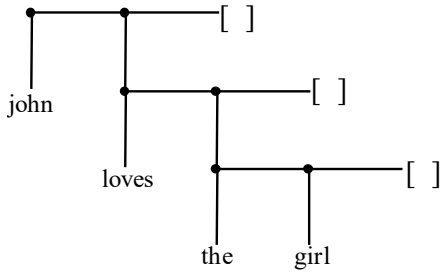


Abb. 4.6. [john, [loves, [the, girl]]

Listen spielen in der Programmierung mit Prolog eine ganz zentrale Rolle. Da dieser Punkt so wichtig ist, soll er in diesem Skript angemessen gründlich und ausführlich behandelt werden. Aus diesem Grund geht es im nächsten Kapitel ausschließlich um die Verarbeitung von Listen in Prolog.