Kapitel 1.
Einleitung: Computerlinguistik und Prolog

1.1. Computerlinguistik

Die Bezeichnung COMPUTERLINGUISTIK hat sich in jüngerer Zeit als Sammelbegriff für eine Vielzahl ganz unterschiedlicher Projekte und Programme etabliert, bei denen es im weitesten Sinne um die Verarbeitung von Sprache und sprachlichen Daten auf Computern geht. Darunter fallen so unterschiedliche Anwendungen wie Synthese und Analyse gesprochener Sprache; Programme zur maschinellen Übersetzung; Programme zur Modellierung sprachlicher Kompetenz auf den verschiedenen sprachlichen Ebenen und so weiter. Für ein Teilgebiet der Computerlinguistik findet sich auch die Bezeichnung *linguistische Datenverarbeitung*, wobei der Computer im wesentlichen als Hilfsmittel zur Verarbeitung von Sprachdaten dient, beispielsweise zur Erstellung von Wortlisten (Konkordanzen,[1] Häufigkeitswörterbücher, rückläufige Wörterbücher[2] etc.) oder zur statistischen Analyse und Beschreibung von sprachlichen Ausdrücken in Texten.

Im vorliegenden Skript ist der Schwerpunkt allerdings anders gelagert: Hier soll die Computerlinguistik als Teildisziplin der Linguistik verstanden werden, deren Ziel die Implementierung linguistischer Theorien oder Teiltheorien auf Computern ist. Ein konkretes Beispiel dafür ist die Implementierung einer Phrasenstrukturgrammatik. In diesem Zusammenhang ist die Computerlinguistik ein wissenschaftlicher Ansatz, der im Spannungsfeld von Disziplinen wie der Linguistik, der Informatik und der künstlichen Intelligenz, der Wissensrepräsentation, den Kognitionswissenschaften usw. angesiedelt ist. Eine genau Trennung ist bei diesen Fachrichtungen nicht möglich, da die Grenzen zwischen ihnen fließend sind. Was nun ist die primäre Zielsetzung der Computerlinguistik in diesem Sinne, welchem Zweck also dient die Umsetzung linguistischer Theorien auf Computern? Im Grunde werden zwei primäre Aufgabenbereiche verfolgt:

- Zum einen geht es um das eher praktische Ziel, auf der Basis von bestimmten Grammatiktypen und -fragmenten Programmsysteme zur maschinellen Sprachverarbeitung zu entwickeln, die zu einem bestimmten Nutzen eingesetzt werden können. Dazu gehören beispielsweise Systeme zur maschinellen Übersetzung, Dialogsysteme, Systeme für die computergestützte Lexikographie, Systeme zur Stil- und Strukturanalyse, automatisches Erstellen von Inhaltsangaben und so weiter. Dieser Aufgabenbereich wird gemeinhin zur ANGEWANDTEN COMPUTERLINGUISTIK gezählt.
- Zum anderen, im Bereich der THEORETISCHEN COMPUTERLINGUISTIK, geht es darum, diejenigen (im wesentlichen algorithmischen)[3] linguistischen Theorien und Modelle, die die Grundlage der Implementierung darstellen, auf ihre Adäquatheit hin zu überprüfen und sie im Sinne der Modellverbesserung zu modifizieren. Hier erfüllt der Computer eine wichtige Funktion, da auch hochgradig komplexe Modelle zu entsprechend aufbereiteten Datenmengen wie beispielsweise Lexika in Beziehung gesetzt und mit diesen gestestet und ausprobiert werden können. Nicht zu vergessen ist bei diesem Ansatz die Tatsache, daß er die Linguisten zwingt, präzise und systematisch zu arbeiten: es ist hier nicht möglich, bestimmte (möglicherweise unbequeme) Probleme im Formalismus zu vernachlässigen oder deren Lösung 'auf später' zu verschieben.

---

[1] Alphabetische Verzeichnisse der (inhaltlich verschiedenen) Wörter in einem Text mit Angabe der Belegstellen.

[2] Alphabetische Verzeichnisse der Wörter einer Sprache, jedoch nach dem Ende eines Wortes; zuerst kommen alle Wörter, die auf den Buchstaben 'a' enden, dann die mit 'b', etc. Sind sind nützlich beispielsweise für morphologische Analysen, denn alle Wörter mit der gleichen Endung finden sich an der gleichen Stelle.

[3] Hier lohnt ein Hinweis auf die Tatsache, daß die Bezeichnung 'algorithmische Linguistik' im Grunde dem entspricht, was durch das engl. *computational linguistics* ausgedrückt wird. Die Bezeichnungen Computerlinguistik und *computational linguistics* sind keine 1–zu–1 Entsprechungen voneinander: *computational* bedeutet soviel wie 'berechenbar' oder eben 'algorithmisch'.

1.2. Prolog

Der Name *Prolog* ist abgeleitet aus *Programming in Logic*. Diese Bezeichnung ist durch die Tatsache begründet, daß die Verarbeitung von Daten in Prolog auf einem Mechanismus (nämlich dem sog. 'Resolutionsverfahren') fußt, der auf einem Teilbereich der Prädikatenlogik basiert. Durch die Eignung für den oben diskutierten. Aufgabenbereich erfreut sich Prolog in Bereichen wie der Künstlichen Intelligenz und der algorithmischen Linguistik weiter Verbreitung. Prolog unterscheidet sich in seiner Arbeitsweise ganz wesentlich von konventionellen Programmiersprachen wie *BASIC*, *C* oder *Pascal*. Letztere sind sog. *imperative* Sprachen (vgl. IMPERATIV = 'Befehlsform'). Sie werden so genannt, weil Programme in diesen Sprachen aus Anweisungsfolgen bestehen, mit denen dem Computer im Detail mitgeteilt wird, welche Aktionen in welcher Reihenfolge auszuführen sind, um ein Problem zu lösen.

Prolog ist im Gegensatz dazu eine *deklarative* (bzw. *prädikative*[4]) Sprache. 'Deklarativ' deshalb, weil ein Programm nicht aus einer Anweisungsfolge, sondern aus einer detaillierten *Beschreibung* (durch "Aussagesätze", vgl. engl. *declarative clause*) des Problems besteht. Programmieren wird bei einer deklarativen (prädikativen) Sprache als Beweisen in einem System von Tatsachen und Schlußfolgerungen aufgefaßt. In Prolog formuliert man sein eigenes Wissen über das Problem, und der Computer versucht, mithilfe dieses Wissens selbständig eine Lösung des Problems zu finden.

Informell könnte man den Unterschied wie folgt darstellen:

Imperativ/Prozedural

1) Führe X aus
2) Führe Y aus
3) Gehe zurück zu 1) usw.

Deklarativ/Prädikativ

X ist wahr.
Y ist wahr.
Wenn X und Y wahr sind, ist auch Z wahr.

In der Sprache der Logik, auf der Prolog basiert, besteht das Wissen aus einer Menge von *wahren Aussagen* und *Regeln*, die wahre Aussagen in wahre Aussagen überführen. Das heißt, das Prolog in der Lage ist, über Regeln aus bekannter Information neue Information abzuleiten. Gibt der Benutzer eine *Behauptung* ein, so versucht das Prolog-System diese auf der Grundlage des verfügbaren Wissens zu beweisen.

Prolog ist aus mehreren Gründen gut für linguistische Zwecke geeignet:

1. Prolog ist primär keine Sprache zur Durchführung von Berechnungen als Manipulation von Zahlen, sondern eine Sprache zur Manipulation von einfachen und komplexen Symbolen; Zahlen sind nur eine besondere Art von Symbolen. Es liegt in der Hand des Anwenders, wie bestimmte Symbole zu interpretieren sind. Beispielsweise ist ein Prolog-Ausdruck wie np(det(der),n(mann)) für Prolog nur ein komplexes Symbol ohne weitere Bedeutung. Im Zusammenhang mit einem Syntaxanalyseprogramm kann dieser Ausdruck jedoch zur Repräsentation der Phrasenstruktur der Wortfolge *der Mann* verwendet werden.
2. Mit Bezug auf die weiter oben angesprochene Zielsetzung der theoretischen Computerlinguistik, also der Implementierung linguistischer Theorien, kann folgendes gelten: Eine Grammatik wird aufgefaßt als Theorie einer Einzelsprache, insofern die Grammatik beispielsweise u.a. spezifiziert, welche Eigenschaften Wortfolgen aufweisen müssen, damit diese als grammatisch gelten können. Eine Grammatik besteht in der Regel aus einer Menge von FAKTEN (beispielsweise über die Zugehörigkeit von Wörtern zu Wortarten: *der* ist ein Artikel, *mann* ist ein Nomen) und REGELN (beispielsweise bezüglich der Zusammensetzung von Sätzen aus Syntagmen: z.B S $\rightarrow$ NP, VP). Dies gilt unabhängig davon, ob man die Grammatik zur Analyse von Ausdrücken oder zu ihrer Synthese (oder Generierung) verwenden will.

[4] Der *Duden "Informatik"* (S. 547) verwendet anstelle von "deklarativ" die Bezeichnung "prädikativ".

Worum es bei Prolog in erster Linie geht, ist also die präzise Umsetzung der der jeweiligen Problemstellung zugrundeliegenden Informationsstrukturen in entsprechende Fakten und Regeln, die Verarbeitung dieser Fakten und Regeln wird dann automatisch vom Programm übernommen und muß nicht, wie bei einem imperativen Programm, schrittweise angegeben werden.

1.2.1. DIE ARBEITSWEISE VON PROLOG

Ein PROLOGPROGRAMM, bzw. allgemein ein LOGIKPROGRAMM, besteht aus einer Menge von Aussagen, welche die Eigenschaften von Objekten und die Beziehungen oder Relationen zwischen diesen Objekten definieren. Die Aussagen eines Prologprogrammes sind entweder Aussagen über individuelle Gegebenheiten (z.B. *Fido ist ein Hund*, *Fido ist größer als Miezie* oder *Das Genus von 'Mädchen' ist Neutrum*), oder Verallgemeinerungen von solchen Aussagen (z.B. *Menschen sind sterblich*, *Hunde sind größer als Katzen*, *eine Sprache mit Verbendstellung weist auch Postposition auf*). Aussagen über individuelle Gegebenheiten werden FAKTEN genannt, die Verallgemeinerungen sind REGELN.

Die Menge der Fakten und Regeln, die ein Logikprogramm ausmachen, sind in einer internen Datenbank gespeichert, die auch WISSENSBASIS genannt wird. Diese Wissensbasis ist sozusagen eine 'Mini-Welt', ein Modell eines Ausschnitts aus der Wirklichkeit.

Die 'Berechnung' eines Logikprogrammes ist die ABLEITUNG der Menge der Aussagen, die aus dem Programm gefolgert werden können. Ein Prologsystem muß also über einen Mechanismus zur Ableitung von Aussagen aus einem Programm verfügen. Dieser Mechanismus basiert, wie schon erwähnt, auf einem spezifischen Beweisverfahren der Prädikatenlogik (worauf an dieser Stelle nicht näher eingegangen werden soll). Man nennt diesen Mechanismus INFERENZMASCHINE, von engl. *inference engine*, wobei Inferenz hier als Schlußfolgerung zu verstehen ist.

Stellt nun ein Benutzer eine entsprechend formulierte Anfrage an Prolog, so wird die Inferenzmaschine aktiv und versucht, diese Anfrage durch die systematische Abarbeitung von Aussagen aus der Wissensbasis zu beweisen und auf diese Art zu einer (oder mehrerer) Antwort(en) zu gelangen. Der Programmablauf bei Prolog kann zunächst wie folgt schematisch dargestellt werden:

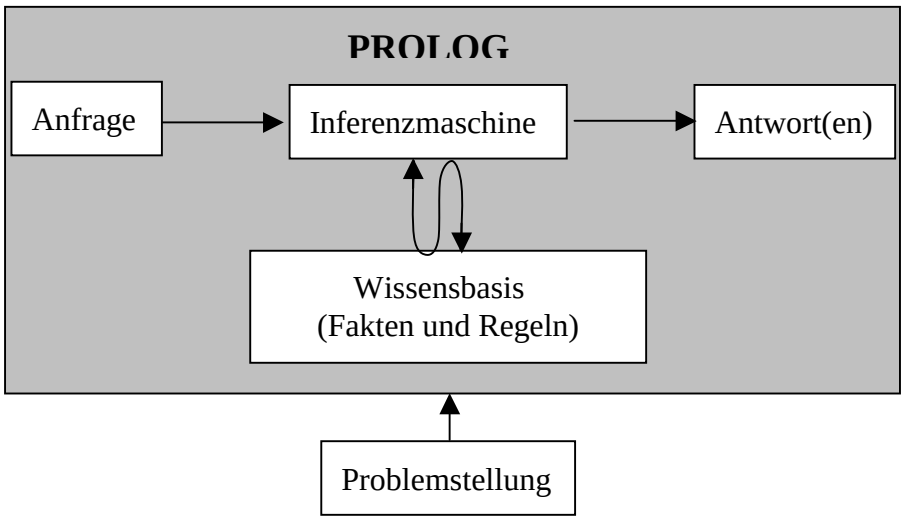


Abb. 1.1. Modell der Prologprogrammierung

Diese Abbildung ist so zu verstehen: eine (beliebige) Problemstellung – z.B. eine Phrasenstrukturgrammatik für das Englische – wird als Menge von Regeln und Fakten in Prolog-Aussagen übersetzt. Dieser Punkt ist die zentrale Aufgabe der Programmierer. Wird eine Anfrage an das Programm gestellt, versucht die Inferenzmaschine auf der Basis eines prädikatenlogischen Algorithmus diese Anfrage durch die Abarbeitung der Wissensbasis zu 'beweisen', um so eine bzw. mehrere mögliche Antworten zu erhalten. Diese Darstellung wird sehr viel anschaulicher, wenn sie im nächsten Kapitel an einem konkreten Beispiel erneut aufgegriffen wird

Dieses Skript soll eine erste elementare Einführung in bestimmte Fragestellungen der theoretischen Computerlinguistik in dem weiter oben spezifizierten Sinne sein. Dabei werden zwei Ziele verfolgt:

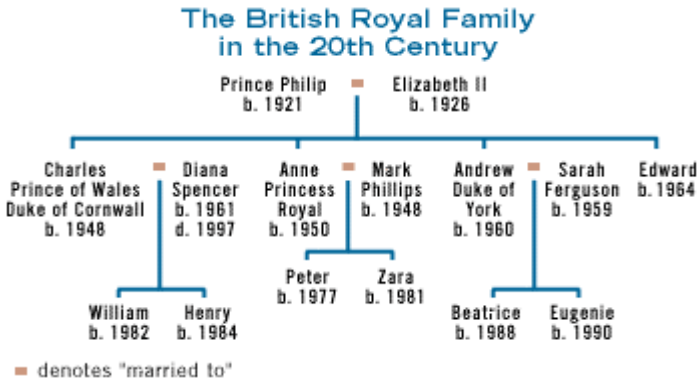
- Einerseits geht es um die Vermittlung von Grundkenntnissen der Programmiersprache Prolog, die sich für die oben genannten Aufgaben gut eignet und in der modernen Computerlinguistik bzw. der Künstlichen Intelligenz recht verbreitet ist.
- Zum anderen geht es darum, anhand konkreter Programmierbeispiele exemplarisch einige der zentralen Probleme und Fragestellungen der theoretischen Computerlinguistik nachzuvollziehen und in der Praxis umszusetzen.

Idealerweise sind diese beiden Zielsetzungen so miteinander verknüpft, daß sich die Aneignung der Prolog-Grundlagen aus den Problemstellungen der computerlinguistischen Implementierungen ergibt. Diesem Anspruch versucht das Skript, so weit es geht, gerecht zu werden. Für die notwendige Sicherheit im Umgang mit Prolog müssen aber auch bestimmte Übungen durchgeführt werden, die mehr oder weniger den Status von Fingerübungen haben und deren Nutzen für die zu erstellenden Programme eher mittelbar ist.

Kapitel 2.
Grundbegriffe: Fakten, Anfragen, Regeln

Anhand eines konkreten (nicht-linguistischen) Gegenstandes wollen wir die ersten Schritte in Prolog angehen, unser erstes kleines Prolog-Programm namens familie.pl verfassen und einige wichtige Grundbegriffe kennenlernen. Die Problemstellung ist so ausgewählt, daß die ihr zugrundeliegende Wissenstruktur völlig einfach nachzuvollziehen ist und wir uns somit darauf konzentrieren können, sie in eine Form zu bringen, die von Prolog verarbeitet werden kann. (Aus diesem Grund findet sich das folgende Beispiel übrigens in fast allen Prolog-Einführungen). Obwohl das Programm zunächst völlig unlinguistisch aussieht, wird in den (Haus)Aufgaben der Tatsache Rechnung getragen, dass die Analyse von Verwandtschaftsbeziehungen und deren Bezeichnung ein weit verbreitetes linguistsches Untersuchungsfeld in der kontrastiven Linguistik ist.

Es geht konkret um einen Familienstammbaum wie den folgenden:



Aus diesem Stammbaum läßt sich eine Vielzahl von unterschiedlicher Information entnehmen, nämlich

1. Information zu Eigenschaften einzelner Personen
2. Information hinsichtlich der Beziehungen, die zwischen einzelnen Personen bestehen.

Zu Punkt eins: Über eine jede Person kann ausgesagt werden, welchen Geschlechts sie ist: Philip ist männlich, Diana ist weiblich usw. und wann sie geboren wurde. (Übrigens: der Stammbaum ist nicht aktuell – auf die Tatsache, dass Familienmitglieder verstorben sind, gehen wir aber nicht ein, und welche Rolle Rittmeister Hewitt in dem Ganzen hatte, wird auch nicht berücksichtigt).

Zu Punkt zwei: Über die Beziehungen zwischen einzelnen Personen können Aussagen gemacht werden: Anne ist mit Mark verheiratet, Sarah ist mit Andrew verheiratet, Charles ist Elternteil von Henry, Peter ist Sarahs Bruder, Eugenie ist Tochter von Andrew, Elisabeth ist Großmutter von Beatrice usw.

Diese und andere Daten sind alle in dem o.a. Stammbaum enkodiert, und wir können sie daraus entnehmen. So gesehen stellt dieser Stammbaum eine Art Mini-Wissensbasis dar. Im folgenden nun soll in kleinen Schritten demonstriert und erklärt werden, wie man diese Daten in genau die Form bringt, die Prolog verarbeiten kann, und wie man somit Anfragen an Prolog stellen kann, die das Programm sinnvoll beantwortet. Es geht also darum, das Wissen so zu kodieren, daß auf eine Anfrage wie z.B. "Wer ist der Großvater von William" die Antwort "Philip" geliefert wird. Dabei soll gezeigt werden, wie man Prolog dazu bringt, mithilfe von Regeln aus einer Reihe von Fakten andere Fakten abzuleiten.

2.1. Fakten

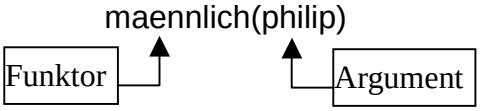
Der erste Schritt besteht darin, Fakten einzugeben. Ein Prolog-Fakt ist eine Feststellung, daß ein Objekt eine bestimmte Eigenschaft hat (*Philip ist männlich*), oder daß zwischen zwei oder mehr Objekten eine bestimmte Beziehung (Relation) besteht (*Elisabeth ist die Mutter von Edward*). Das bedeutet, daß Fakten Aussagen über Eigenschaften von Einzelindividuen oder Beziehungen zwischen Einzelindividuen sind. Dazu gehören in unserem Stammbaum beispielsweise die Geschlechtszugehörigkeit der einzelnen Personen.

Die Geschlechtszugehörigkeit der Personen im Stammbaum

Umgangsprachlich würde man das so ausdrücken: Philip ist männlich, Elizabeth ist weiblich, Charles ist männlich, Diana ist weiblich usw. Leider ist Prolog nicht in der Lage, umgangssprachliche Ausdrücke zu verarbeiten. Eine Aussage wie "Philip ist männlich" muß erst in eine andere Form gebracht werden, um von Prolog verarbeitet werden zu können. 'Männlich' drückt die Eigenschaft oder das Attribut aus, welches dem Individuum Philip zugeschrieben wird. In Prolog wird dieses Fakt auf die folgende Weise notiert:

maennlich(philip).

Das heißt, daß der Ausdruck "Philip ist männlich" in eine eine Funktor-Argument-Struktur übersetzt wird, in der die Eigenschaft oder das Attribut den Funktor stellt und das Individuum das Argument dieses Funktors. Der Funktor wird auch als 'Prädikatsname' bezeichnet, dazu mehr s.u. Der Funktor steht auf der linken Seite, das Argument steht in Klammern rechts daneben. Zu achten ist dabei auf die Groß/Kleinschreibung: es werden hier nur Kleinbuchstaben verwendet:



Analog verfahren wir bei den Geschlechtszugehörigkeit der anderen Personen: "Elizabeth ist weiblich" wird in Prolog durch weiblich(elisabeth) repräsentiert; "Andrew ist männlich" durch maennlich(andrew) usw.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 1 - 3** am Kapitelende angehen

B) Die Beziehungen zwischen einzelnen Personen im Stammbaum

In diesem Abschnitt geht es nicht um Eigenschaften einzelner Personen, sondern um die Beziehungen zwischen jeweils zwei Personen. Wir betrachten zunächst nur die Elternteil- und die Ehegatten-Beziehung, da wir, wie wir weiter unten sehen werden, alle anderen Verwandschaftsbeziehungen (Vater, Mutter, Schwester, Tante usw. usf.) aus diesen beiden ableiten können.

Auch hier beginnen wir damit, die einzelnen Beziehungen zunächst umgangssprachlich zu formulieren, also z.B. "Philip ist Elternteil von Anne". Dieser Satz muß, analog zu dem Satz "Philip ist männlich", auch wieder in eine Funktor-Argument-Struktur übersetzt werden, um von Prolog verarbeitet werden zu können. Er unterscheidet sich aber von "Philip ist männlich" dadurch, daß hier nicht eine Eigenschaft des Einzelindividuums Philip dargestellt ist, sondern die Beziehung, die zwischen den Einzelindividuen Philip und Anne besteht. In diesem Fall fungiert die jeweilige Beziehung als Funktor, welcher dann nicht ein, sondern zwei Argumente hat: elternteil(philip,anne). Genauso verfährt man bei "Elizabeth ist Elternteil von Charles": elternteil(elisabeth,charles). Dabei ist zu beachten, daß die einzelnen Argumente jeweils durch Kommata voneinander getrennt werden.

Ein wichtiger Punkt noch: die Frage nach der Reihenfolge der Argumente. Diese wird nämlich von dem jeweiligen Programmierer festgelegt. Im obigen Fall steht zuerst das Elternteil, danach das Kind, und das entspricht auch in etwa dem umgangssprachlichen Satz "Elisabeth ist Elternteil von Charles", in welchem auch zuerst das Elternteil und dann das Kind genannt ist. Etwas anders verhält es sich bei der Ehegatten-Beziehung: diese ist symmetrisch, d.h. man könnte sowohl "Charles ist Ehegatte von Diana" als auch "Diana ist Ehegatte von Charles" sagen, also könnte das erste Argument entweder der Ehemann oder die Ehefrau sein. In einem solchen Fall liegt es im Ermessen des Programmierers, die Reihenfolge zu bestimmen. Wenn man sich allerdings einmal festgelegt hat, muß die Reihenfolge konsequent durchgehalten werden. Wird also die Ehegatten-Beziehung in Prolog durch die folgende (abstrahierte) Struktur ehegatte(Frau,Mann) eingeführt, kann man nicht an späterer Stelle die Reihenfolge umdrehen und ehegatte(Mann,Frau) daraus machen.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 4 - 7** am Kapitelende angehen

2.2. Anfragen

Neben den Fakten sind die Anfragen die zweite Form der "Aussage" in einem Logikprogramm. Anfragen sind ein Mittel zur Gewinnung von Information aus Logikprogrammen. Während ein Fakt feststellt, daß ein Objekt eine bestimmte Eigenschaft hat oder daß eine bestimmte Beziehung zwischen Objekten besteht, fragt eine Anfrage danach, ob es zutrifft, daß ein Objekt eine bestimmte Eigenschaft hat oder eine bestimmte Beziehung zwischen Objekten besteht.[5] Auf unser Stammbaum-Beispiel bezogen: eine mögliche Anfrage wäre z.B. *Trifft es zu, daß Elizabeth ein Elternteil von Andrew ist ?*:

?- elternteil(elizabeth,andrew).

Da wir diese Aussage als Fakt in die Wissensbasis haben einlesen lassen, lautet die entsprechende Prolog-Antwort:

Yes

Weitere möglich Fragen an unsere Wissensbasis wären z.B.

?- weiblich(sarah).

Yes

?- elternteil(anne, zara).

Yes

Die Anfrage

?- elternteil(henry, anne).

würde allerdings als Antwort

No

erhalten — es ist klar, warum: diese Anfrage läßt sich nicht über ein Fakt in der Wissenbasis beweisen, da es ein solches Fakt gar nicht gibt.

All diese Anfragen beziehen sich ganz konkret auf bestimmte Einzelindividuen. Was aber ist zu tun, wenn wir beispielsweise nicht wissen wollen, ob Charles mit einer bestimmten Person Diana verheiratet ist, sondern ob er überhaupt verheiratet ist, und gegebenenfalls mit wem? In diesem Falle muß die Anfrage mit einer VARIABLEN formuliert werden, die für einen bestimmten, aber noch nicht bekannten Wert steht.

?- verheiratet(charles, X).

Diese Anfrage veranlaßt Prolog, in der Wissensbasis 'nachzugucken', ob es einen Wert für die Variable X gibt, welcher die Aussage verheiratet(charles, X) wahr macht. Einen solchen Wert gibt es in der Tat, entsprechend lautet die Antwort

X=diana

Das heißt, Prolog gibt als Antwort nicht nur an, ob die Anfrage zutrifft oder nicht, sondern auch welcher Wert für die Variable die Aussage wahr macht. Die Schreibmarke bleibt in diesem Falle hinter dem letzten ausgegebenen Zeichen stehen. Durch Eingabe eines Semikolons ';' wird das System veranlaßt, nach weiteren Lösungen zu suchen. Werden keine weiteren Lösungen gefunden (was bei diesem Beispield der Fall ist), erscheint die Meldung No.

VARIABLE in Logikprogrammen (LOGISCHE VARIABLE) unterscheiden sich grundlegend von Variablen in imperativen Sprachen.[6] Eine logische Variable ist ein Name, der im Zuge der Berechnung bestimmten Objekten zugeordnet werden kann. Man sagt dann, die Objekte seien an die Variable GEBUNDEN. Diese Verwendung von Variablen läßt sich mit der Funktion von Pronomina in natürlichen Sprachen vergleichen.

---

[5] Wir werden später sehen, daß Fakten und Anfragen nur Sonderfälle von Regeln sind.

[6] In einer Sprache wie z.B. Pascal ist eine Variable ein reservierter Speicherplatz, an dem durch eine explizite WERTZUWEISUNG ein Wert eines bestimmten Typs gespeichert werden kann. Der Speicherinhalt einer Variablen bleibt solange erhalten, bis er durch eine erneute Wertzuweisung verändert worden ist.

Syntaktisch werden Variable durch Zeichenketten bezeichnet, die mit einem Großbuchstaben oder einem Unterstreichungsstrich beginnen.

Beispiele für die Schreibweise von Variablen:

Nominalphrase, NP, Np, X, _singt usw.

Variable, die mit dem Unterstreichungsstrich '_' beginnen, sind sog. “anonyme” Variable, die wir verwenden können, wenn es im Zusammenhang auf den Wert der Variablen nicht ankommt. Die Bindung an einen konkreten Wert wird bei der Verwendung einer anonymen Variablen nicht ausgegeben. Um beispielsweise herauszufinden, ob Charles verheiratet ist, kann die Anfrage lauten

?- verheiratet(charles,_gattin).
Yes

An dieser Stelle lohnt ein Hinweis darauf, daß bei den anonymen Variablen zu differenzieren ist zwischen einer Variable, die mit einem Unterstrich beginnt (z.B. _X, _ehegatte, _xyz usw.) und dem einfachen Unterstrich '_'. Am besten wird der Unterschied an einem Beispiel verdeutlicht: die komplexe Suchanfrage (siehe dazu mehr weiter unten)

?- maennlich(_), ehegatte(charles,_).

wird als Ergebnis die Ausgabe

Yes

bekommen - die anonyme Variable '_' ist in diesem Fall natürlich an verschiedene Werte gebunden, d.h. sie hat unterschiedliche Referenz. Im Gegensatz dazu die Anfrage:

?- maennlich(_person), ehegatte(charles,_person).

Hier lautet die Antwort

No.

Der Grund: Bei der anonyme Variable '_person' ist es zwar egal, welchen Wert sie erhält, aber dieser muß für beide Beweisziele identisch sein. Da wir in unserem Stammbaum aber die Ehefrauen als zweites Argument aufgeführt haben, wird es ausgeschlossen sein, einen Wert zu finden, der sowohl als Argument von maennlich/1 als auch als zweites Argument von ehegatte/2 auftritt.

2.3. Regeln

In unserer Prolog-Wissensbasis ist nun dargestellt, welche Personen im Stammbaum männlich oder weiblich sind, wer mit wem verheiratet ist und wer Elternteil von wem ist. Wir können entsprechende Anfragen an das Programm stellen. Es stecken aber noch mehr Informationen in dem Stammbaum, z.B.: wer ist Mutter von wem, wer ist Enkel von wem, welche Personen sind miteinander verschwägert usw. Um auch diese Daten in unsere Wissensbasis zu bekommen, könnten wir theoretisch so weitermachen wie gehabt: wir geben lauter Fakten ein wie z.B. mutter(elisabeth,charles) oder enkel(henry,philip) oder bruder(peter,zara).

Es gibt aber eine Lösung, die wesentlich eleganter (und viel weniger schreibintensiv) ist: Man leitet diese Beziehungen aus den bereits eingegebenen Beziehungen und Eigenschaften der Personen ab. 'Ableiten' heißt in diesem Fall, daß wir nicht mehr Aussagen über Einzelindividuen machen, sondern auf der Basis der bereits bekannten Fakten allgemeingültige Regeln formulieren.

Als Beispiel betrachten wir uns die Beziehung kind_von. Es wäre ziemlich mühsam, alle kind_von Beziehungen in der Form von Fakten wie kind(Kind,Elternteil)[7] einzugeben. Außerdem, und das ist der entscheidende Punkt, wissen wir ja, daß zwischen der Beziehung kind_von und der Beziehung elternteil_von ein systematischer Zusammenhang besteht, insofern die kind_von-Beziehung gewissermaßen die Umkehrung der elternteil_von-Beziehung. ist, und diese ist ja bereits in der

[7] Die Argumente dieses Ausdrucks sind die beiden Metavariable 'Kind' und 'Elternteil'. An dieser Stelle könnte natürlich auch einfach kind(K,E) oder kind(X,Y) stehen. Durch die Verwendung dieser Variablen aber wird gleich ein Hinweis auf den Charakter bzw. die Art des Argumentes gegeben. Es bietet sich übrigens immer an, auch bei der Programmierung sog. 'sprechende' Variable zu verwenden, d.h. Variable, die mehr oder weniger selbst dokumentieren , wofür sie stehen. Das fördert die Lesbarkeit von Programmen ungemein.

Prolog-Wissensbasis. Das wollen wir uns zunutze machen, indem wir die folgende Regel formulieren: *Eine beliebige Person K ist Kind von einer beliebigen Person E falls die Person E Elternteil von der Person K ist*. Diese umgangssprachliche Äußerung ist klar. Wie wir allerdings sehen, geht es hier nicht mehr um bestimmte Einzelpersonen, also Charles, Andrew und so weiter, sondern um eine allgemeingültige Aussage. Für das Programm bedeutet das soviel wie: "Wenn wahr ist, daß E Elternteil von K ist, dann ist auch wahr, daß K Kind von E ist." Da die elternteil_von Beziehungen unseres Stammbaumes gänzlich in die Prolog-Wissensbasis eingegeben wurden, kann Prolog leicht überprüfen, für welche Personen die kind_von Beziehung zutrifft. Die Frage ist nur, wie man einen solchen Ausdruck in Prolog übersetzt. Dazu betrachten wir uns den Satz nochmal zeilenweise:

1. *Eine beliebige Person K ist Kind von einer beliebigen Person E*

2. *falls*

3. *die Person E Elternteil von der Person K ist.*

In der ersten Zeile wird ausgesagt: eine beliebige Person K ist Kind von einer beliebigen Person E. Hier handelt es sich um die kind_von Beziehung zwischen zwei Personen, allerdings wird hier keine Aussage über ein Einzelindividuum gemacht, sondern eine Aussage, die allgemeine Gültigkeit hat. Die Notation in Prolog sieht für diesen Fall vor, daß die Argumente in der Funktor-Argument-Struktur als Variable wiedergegeben werden. Diese sind durch Großbuchstaben gekennzeichnet. Die erste Zeile würde also wie folgt wiedergeben:

1. kind(K,E).

In der zweiten Zeile finden wir das Wort *falls*. Dieses ist in Prolog durch Doppelpunkt/Bindestrich repräsentiert (der übrigens eine abstrahierte Form des umgekehrten Implikationspfeiles aus der Logik ('←') darstellt):

2. :–

In der dritten Zeile schließlich findet sich wieder die Beziehung elternteil_von, allerdings auch wieder nicht für Einzelindividuen formuliert:

3. elternteil(E,K).

Von äußester Bedeutsamkeit ist die Tatsache, daß die Variablen der dritten Zeile in engem Bezug zu denen der ersten Zeile stehen – es macht auch umgangsprachlich wenig Sinn, zu sagen: Eine Person K ist Kind von einer Person E falls die Person X Elternteil von der Person Y ist. Die kind_von Beziehung ist, wie gesagt, die Umkehrung der elternteil_von Beziehung, es müssen also dieselben Variablen verwendet werden, damit Prolog die richtigen Zuordnungen treffen kann. Unser umgangsprachlicher Ausdruck hat in Prolog demnach die folgende Form:

Umgangsprachlich	**Prolog**
Eine beliebige Person K ist Kind von einer beliebigen Person E	kind(K,E)
falls	:–
die Person E Elternteil von der Person K ist.	elternteil(E,K).

Der Prolog Ausdruck wird in eine Zeile geschrieben:
kind(K,E):- elternteil(E,K).
Damit ist die kind_von Beziehung aus der elternteil_von Beziehung abgeleitet. Bei der Erstellung von Regeln ist es extrem wichtig, zu prüfen, welche Information vorhanden ist, um sich dann zu überlegen, wie man davon ausgehend neue Information produziert. Dabei sollte man sich immer zunächst umgangsprachlich klarmachen, wie die Dinge eigentlich liegen.

Dazu das nächste Beispiel: wir wollen eine Regel für die mutter_von Beziehung schreiben. Zuerst formulieren wir wieder umgangssprachlich auf der Basis des Bekannten, was eine Mutter eigentlich ist: *Eine beliebige Person M ist Mutter von einer beliebigen Person K falls die Person M weiblich ist und die Person M Elternteil von der Person K ist.* Die Konjunktion *und* wird in Prolog durch ein Komma repräsentiert, so daß wir diesen Satz zeilenweise wie folgt in Prolog übersetzen:

Umgangsprachlich	Prolog
Eine beliebige Person M ist Mutter einer beliebigen Person K	mutter(M,K)
falls	:–
die Person M weiblich ist	weiblich(M)
und	,
die Person M Elternteil von der Person K ist.	elternteil(M,K).

Die mutter_von Beziehung hat in Prolog also diese Form:
mutter(M,K):- weiblich(M),elternteil(M,K).

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 8 und 9a** am Kapitelende angehen

Kommen wir zu einem weiteren Beispiel – die Beziehung grossmutter/2. Erneut machen wir uns zunächst umgangssprachlich klar, wie man diese Beziehung durch eine allgemeine Regel erfassen kann. Eine mögliche Formulierung lautet: *Eine beliebige Person G ist Großmutter einer beliebigen Person K, falls die Person G weiblich ist und Elternteil einer Person E, die ihrerseits Elternteil der Person K ist.*

Umgangsprachlich	Prolog
Eine beliebige Person G ist Großmutter einer beliebigen Person K	grossmutter(G,K)
falls	:–
die Person G weiblich ist	weiblich(G)
und	,
die Person G Elternteil einer beliebigen Person E ist	elternteil(G,E).
und	,
die Person E Elternteil der Person K ist.	elternteil(E,K).

Zeilenweise:

grossmutter(G,K):– weiblich(G), elternteil(G,E), elternteil(E,K).

Bevor wir noch weitere Verwandschaftsbeziehungen in Form von Regeln eingeben, erfolgen erst einige grundsätzliche Informationen über Regeln im allgemeinen.

Die allgemeine Form einer Regel

Eine Regel besteht normalerweise aus einem Regelkopf und einem Regelrumpf:

mutter(M,K)	:-	weiblich(M), elternteil(M,K).
Regelkopf		Regelrumpf

verallgemeinert:

a	:-	b_1, b_2, : . . b_n.

wobei a und alle b *Beweisziele* sind. a ist der Kopf der Regel, b_1, b_2 - b_n bilden den Rumpf.

Ein solcher Ausdruck ist eine KLAUSEL (im Grunde genommen eine Hornklausel, siehe dazu das Skript zum Grundkurs *Mathematische und logische Grundlagen der Linguistik*).

Die Regel für mutter/2 besagt, daß der Kopf mutter(M,K) erfüllbar ist, wenn die Bedingungen im Rumpf insgesamt erfüllbar sind, wenn sich also sowohl weiblich(M) als auch elternteil(M,K) für zwei Werte von M und K "beweisen" lassen – z.B. durch das Auffinden entsprechender Fakten in der Wissensbasis.

Fakten und Anfragen können nunmehr als Sonderfälle von Regeln aufgefaßt werden. Ein Fakt ist ein Prädikat, das immer erfüllbar ist. Das ist gleichbedeutend mit einer Regel, deren Rumpf immer erfüllt

wird. In der Tat gibt es in Prolog ein vordefiniertes Prädikat, das immer erfüllt wird. Es handelt sich um das argumentlose true/0. Eine Anfrage mit true/0 ist immer erfolgreich.[8]

?- true.

Yes

Ein Fakt wie weiblich(diana) ist gleichbedeutend mit einer Regel, deren Rumpf true ist: weiblich(diana) **:-** true.

Der Rumpf kann in diesem Fall per Konvention weggelassen werden. In der internen Wissensbank werden jedoch alle Fakten als Regeln behandelt, deren Rumpf aus dem Prädikat true besteht. Man kann dies leicht mit einem vordefinierten Systemprädikat prüfen, das einen direkten Zugriff auf diese Wissensbank erlaubt. Es handelt sich um das Prädikat clause/2, dessen erstes Argument für den Kopf einer Regel steht und dessen zweites für den Rumpf. Vorausgesetzt der Fakt weiblich(diana) ist in der Datenbank enthalten, erhalten wir folgendes:

?- clause(weiblich(diana), Rumpf).

Rumpf = true

Eine Anfrage ist ebenfalls eine Sonderform einer Regel: diese besteht nur aus dem Rumpf.

Prädikat

Ein wichtiger Begriff soll in diesem Zusammenhang eingeführt werden, nämlich der Begriff PRÄDIKAT. Zunächst erfolgt die Definition, dann die Erläuterung:

> **Prädikat**
>
> *Ein Prädikat ist die Menge aller Klauseln, deren Köpfe denselben Funktor (auch: Prädikatsnamen) und die gleiche Stelligkeit haben.*

Alle Prolog-Ausdrücke, die wir kennengelernt haben, also die Regeln einerseits und die Fakten und die Anfragen als Sonderformen der Regel andererseits, haben die Form von KLAUSELN. Die Stelligkeit einer Klausel bezieht sich auf die Zahl der Argumente, die die Klausel hat: maennlich(charles) hat ein Argument, nämlich 'charles', und ist einstellig. ehegatte(philip,elizabeth) hat zwei Argumente, nämlich 'philip' und 'elizabeth', ist also zweistellig, wie beispielsweise auch kind(K,E). Alle Klauseln, die denselben Funktor und dieselbe Stelligkeit haben, bilden ein Prädikat. Die korrekte Schreibweise dafür gibt den Prädikatsnamen (also den Funktor) und, dahinter durch einen Schrägstrich abgeteilt, die Stelligkeit wieder. Die Prädikate in der Datei familie.pl lauten also maennlich/1, weiblich/1, ehegatte/2, kind/2 usw. Die Grenze der Argumentzahl ist übrigens nach oben offen (bei zuvielen Argumenten verliert man allerdings rasch den Überblick); es gibt aber auch nullstellige Prädikate (dazu mehr weiter unten).

Der Gültigkeitsbereich von Variablen

Der Gültigkeitsbereich einer Variablen ist die Regel, in der sie vorkommt, d.h. gleichnamige Variablen, die in verschiedenen Regeln vorkommen, werden als verschiedene Variable behandelt. Betrachten wir dazu die beiden folgenden Regeln:

1. kind_von(P1,P2) :– elternteil(P2, P1).
2. mutter(P1,P2) :– weiblich(P1), elternteil(P1,P2).

Die Variablen P1 und P2 kommen sowohl in Regel 1 als auch in Regel 2 vor. In Regel 2 taucht die Variable P1 zweimal auf: in weiblich(P1) und elternteil(P1,P2). Hier muß tatsächlich Identität vorliegen. Hingegen handelt es sich in kind_von(P1,P2) (in Regel 1) und mutter(P1,P2) (in Regel 2) trotz des gleichen Namens um verschiedene Variable – ihr jeweiliger Gültigkeitsbereich endet bei der individuellen Regel, also bei dem Punkt.[9]

---

[8] Das Gegenstück zu true/0 ist fail/0, eine Prädikat, das immer fehlschlägt. Zur Notation (Funktor/n) siehe den Abschnitt über Prädikate

[9] Um einen Namenskonflikt zu vermeiden, werden bei Verwendung beider Regeln in einem tatsächlich Beweis die namensgleichen Variablen von Prolog so umbenannt, daß sie nicht mehr verwechselt werden können.

Typen von Anfragen

- *Anfragen zur Verifizierung eines Sachverhalts*. Sie verhalten sich wie Entscheidungsfragen (*ja*/*nein*-Fragen), die fragen, ob ein bestimmter Sachverhalt zutrifft oder nicht. Sie enhalten keine ungebundenen (freien) Variablen:
  ?- ehegatte(andrew, sarah).
  fragt, ob Andrew mit Sarah verheiratet ist.
- *Suchanfragen* Diese fragen nach Werten, die an eine oder mehrere Variable "gebunden" werden sollen. Sie entsprechen den Wortfragen. Sie fordern das System dazu auf, einen Wert für eine oder mehrere ungebundene Variable zu finden. Die Anfrage
  ?-elternteil(X,william).
  fragt, wer Elternteil von William ist. Die Antwort wird an die Variable X gebunden.
- *Anfragen zur Ausführung einer Aktion, Befehle*. Sie fordern das System auf, eine bestimmte Operation auszuführen:
  ?- consult(familie).
  ist eine Aufforderung, eine Datei namens familie zu lesen und die darin enthaltenen Fakten und Regeln der aktuellen Wissensbasis hinzuzufügen.

Komplexe Anfragen

Mehrere Anfragen können logisch durch *und* oder *oder* verknüpft werden. Man kann also den Antwortbereich einschränken, indem man überprüfen läßt, ob mehrere Bedingungen gleichzeitig zutreffen. Die *und*-Verknüpfung wird durch ein Komma, die *oder*-Verknüpfung durch ein Semikolon ausgedrückt. Die Anfrage

?- elternteil(Person,_) , weiblich(Person).

fragt nach allen Personen in der Wissensbasis (Variable Person), die zwei Bedingungen erfüllen:

1. Sie müssen Elternteil von irgend jemandem sein (anonyme Variable _).
2. Sie müssen weiblichen Geschlechts sein.

Mit anderen Worten, es wird nach allen Müttern gefragt.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgabe 9b** am Kapitelende angehen

Mit diesem Wissen wollen wir erneut auf das Modell der Prologprogrammierung aus dem ersten Kapitel zurückkommen, welches nun wie folgt konkretisiert werden kann:

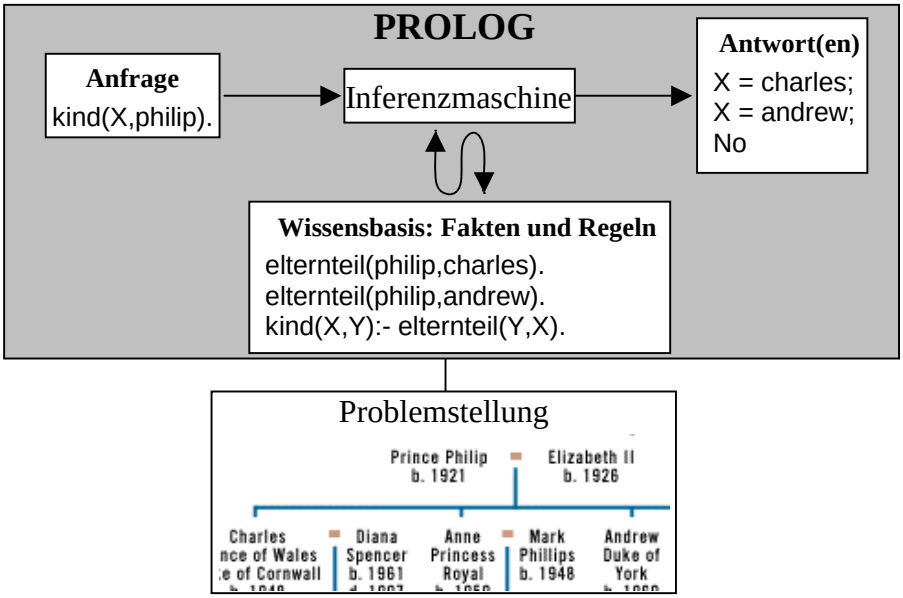


Abb. 1.2. Modell der Prologprogrammierung am Beispiel 'Stammbaum'

Diese Graphik ist nun gut nachzuvollziehen. Die 'Problemstellung' ist in diesem Fall der Stammbaum der englischen Royals. Die in diesem enthaltene Information ist in eine Reihe von Prolog-Fakten und Regeln übersetzt worden, welche in die Prolog-Wissensbasis eingelesen worden sind. Wie wir gesehen haben, ist dieser Vorgang die zentrale Aufgabe der Programmierer. Grundvoraussetzung für eine erfolgreiche Übersetzung ist, daß man sich zunächst Klarheit verschafft über die Art und Weise, die Problemstellung in Aussagen zu erfassen, aus denen durch entsprechende Regeln möglichst ökonomisch andere Aussagen abzuleiten sind. Eine Anfrage, wie in Abbildung 1.2. kind(X,philip), aktiviert die Prolog-Interferenzmaschine, welche systematisch die Wissensbasis durchforstet, um eine geeignete Antwort zu finden. In diesem Fall geht es um die verschiedenen Möglichkeiten, die Variable X an einen Wert zu binden um so die Anfrage zu beweisen. In der Wissensbasis der Abbildung 1.2. bieten sich dafür genau zwei Möglichkeiten an, die entsprechenden Variablenbinddungen werden als Antwort ausgegeben.

2.4. Faustregeln zur Übersetzung von natürlichsprachigen Ausdrücken in Prolog.

Zum Abschluß der zweiten Kapitels sollen nun noch einige Faustregel zur Überführung natürlich-sprachlicher Ausdrücke in Prolog-Ausdrücke erfolgen. Wie genau die sprachlichen Mittel aussehen, mit denen Prolog arbeitet, also eine genauere Spezifikation der formalen Seite dieser Programmier-sprache, folgt weiter unten.

Wir haben gesehen, daß ein Prologprogramm aus einer Menge von Klauseln besteht, die entweder Fakten wiedergeben, d.h. Einzelaussagen über Eigenschaften von Objekten bzw. Beziehungen zwischen Objekten einerseits, und Regeln, d.h. Aussagen über Eigenschaften von Klassen von Objekten bzw. Beziehungen zwischen Klassen von Objekten, die gelten, falls bestimmte Bedingungen erfüllt sind andererseits.

Es ist allerdings nicht immer ganz einfach, Ausdrücke der Alltagssprache, mit welchen wir bestimmte Sachverhalte beschreiben würden, in äquivalente Prologklauseln zu übersetzen.

In dem Satz *Das Pferd ist im (= in dem) Stall* ist einigermaßen klar, daß mit *das Pferd* ein bestimmtes Individuum gemeint ist und mit *dem Stall* ein bestimmter Ort. Es handelt sich in beiden Fällen um Objekte, die wir durch Konstante (z.B. Atome, s.u.) ausdrücken, und die in der Klausel als Argumente vorkommen werden. Die Präposition *in* stellt eine Beziehung zwischen diesen Objekten her, wird also durch einen Funktor in einer Funktor-Argument-Struktur wiederzugeben sein. Ein möglicher Prologausdruck dafür ist also in(pferd, stall).

In dem Satz *Das Pferd ist ein Säugetier* bezeichnet der Ausdruck *das Pferd* nicht ein einzelnes Individuum, sondern eine Klasse, nämlich die Klasse der Pferde im Allgemeinen. Ebenso meint *ein Säugetier* nicht ein individuelles Tier sondern Säugetiere im Allgemeinen. Zwischen beiden besteht eine allgemeine, “regelhafte” Beziehung: jedes Ding, das ein Pferd ist ist auch ein Säugetier. *Pferd* und *Säugetier* sind hier Allgemeinbegriffe und werden als Prädikate wiedergegeben: pferd(X) bzw. saeugetier(X). Der Satz gibt eine Regel wieder: saeugetier(X) :- pferd(X).

Im folgenden werden einige “Faustregeln” zur Übersetzung von alltagsprachlichen Ausdrücken in Prolog gegeben:

- Eigennamen wie *Bart* oder *Lisa* werden zu Konstanten: bart, lisa. Symbolische Konstante werden üblicherweise durch Atome wiedergegeben. Sie können aber auch durch Zeichenketten oder allgemein durch Strukturen ausgedrückt werden (zu den Begriffen ATOM, ZEICHENKETTE und STRUKTUR später mehr).
- Substantive wie *Katze*, oder *Tisch*, die Allgemeinbegriffe bezeichnen, werden zu einstelligen Prädikaten: katze(ophelia),tisch(tisch25).

- Substantive wie *Vater, Kind*, oder *Schwester*, die Beziehungen bezeichnen, werden zu zweistelligen Prädikaten: vater(homer, bart), kind(lisa, marge), schwester(lisa, maggie).[10]
- Bei generischen Substantiven wie Ding, Sache, Umstand können auch Variable gemeint sein. Gleiches gilt für Pronomina. *Wer rastet, der rostet* wäre als rostet(Person):-rastet(Person) wiederzugeben.
- Adjektive wie *rot* oder *bescheiden*, die Eigenschaftsbegriffe bezeichnen, werden zu einstelligen Prädikaten: rot(zora), bescheiden(lisa). Bei Ausdrücken wie *Fredi ist ein weißer Elephant* ist zu bedenken, daß durch die Nominalphrase zwei Eigenschaften ausgedrückt werden: weiss(fredi) und elephant(fredi). *Waldi ist dumm, faul, und gefräßig* ist eine Zusammenfassung von 3 Einzelfakten:
  dumm(waldi).
  faul(waldi).
  gefraessig(waldi).
- Die Komparative von Eigenschaftswörtern bezeichnen Relationen und sind demgemäß als mehrstellige Prädikate darzustellen: kleiner(lisa, bart), schwerer(blei, eisen).
- Es gibt auch Adjektive, die Beziehungen bezeichnen, z.B. *abhängig, unterlegen, zuträglich:* unterlegen(bart,lisa)
- Intransitive Verben wie *spazierengehen* oder *schlafen* werden zu einstelligen Prädikaten: spazierengehen(claudio), schlafen(theodor).
- Transitive Verben wie *sehen* oder *tragen* werden zu zweistelligen Prädikaten: sehen(bart, lisa), tragen(marge, kleid).
- Bitransitive Verben wie *geben* werden zu dreistelligen Prädikaten: geben(lisa, bart, buch) ist die Wiedergabe von *Lisa gibt Bart ein Buch*.
- Bei Passivsätzen kommt es darauf an, ob das Agens (das, was im Aktiv als Subjekt erscheint) ausgedrückt wird oder nicht. Wird das Agens ausgedrückt, bietet es sich an, den Satz ins Aktiv zu transformieren und dann zu übersetzen, z.B. *Bart wird von Lisa unterstützt = Lisa unterstützt Bart* → unterstuetzt(lisa, bart). Wird das Agens nicht ausgedrückt, verhält sich das Verb wie ein intransitives Verb. Am einfachsten verwendet man das Partizip als Prädikat: *die Socken werden gewaschen* → gewaschen(socken).
- Das Hauptverb wird gewöhnlich zum Kopf einer Regel, die übrien Bestandteile gehören zum Rumpf.
- Das Verb *sein* in seinen verschiedenen Formen hat gewöhnlich keine Prologentsprechung, statt dessen wird das darauf folgende Substantiv, Adjektiv, oder Partizip zum Hauptprädikat: *Homer ist ein Idiot*: idiot(homer).
- Präpositionen bezeichnen Relationen (zwei- und mehrstellig): auf(buch, tisch) für *Das Buch liegt (ist, steht, befindet sich) auf dem Tisch* in(anzug, schrank)*: der Anzug ist im Schrank*; zwischen(bremen, frankfurt, hamburg) für *Bremen liegt zwischen Frankfurt und Hamburg*.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgabe 10** am Kapitelende angehen

[10]Solche Substantive können jedoch auch als Namen *(der Vater, das Kind, die Schwester*, z.B. in einem (nicht besonders geglückten) Ausdruck wie 'Die Mutter liebt den Vater': liebt(mutter,vater) oder als einstellige Prädikate verwendet werden (*Hans ist Vater geworden:* vater(hans)).

Übungen zu Kapitel 2

Aufgabe l: Übersetzt (auf Papier) die folgenden Sätze in geeignete Prolog-Fakten.
Charles ist reich.
Sarah ist rothaarig.
Henry ist minderjährig.
Anne ist geschieden.

Aufgabe 2: Übersetzt (auf Papier) die folgenden Prolog-Fakten in umgangsprachliche Sätze:
moppelig(andrew).
unverheiratet(edward).
klein(elizabeth).

Aufgabe 3: Legt eine Datei familie.pl und gebt in diese die Fakten über die Geschlechts-zugehörigkeit der einzelnen Personen des Stammbaumes ein

Aufgabe 4: Übersetzt (auf Papier) die folgenden Sätze in geeignete Prolog-Fakten:
Charles liebt Polo.
Eugenie und Beatrice sind Geschwister.
Sarah schreibt Kinderbücher.

Aufgabe 5: Übersetzt (auf Papier) die folgenden Prolog-Fakten in umgangssprachliche Sätze:
liiert_mit(charles,camilla).
besitzt(elizabeth,hunde).
beruf(andrew,pilot).

Aufgabe 6: Warum wird es bei den folgenden Prolog-Fakten (mit Bezug auf den konkreten Stammbaum) Komplikationen geben?
elternteil(philip,charles).
elternteil(elisabeth,andrew).
elternteil(anne,zara).
elternteil(henry,diana).
elternteil(andrew,beatrice).

Aufgabe 7: Vervollständigt in der Datei familie.pl die Daten über die Elternteil_von- und die Ehegatten-Beziehungen der einzelnen Personen des Stammbaumes. Laßt anschließend den Inhalt der Datei mit consult/1 in die Prolog-Wissensbasis einlesen.

Aufgabe 8: Wie würdet Ihr (auf Papier) die folgenden Ausdrücke in Prolog wiedergeben:
Magda mag Max
Mädchen mögen mollige Männer

Aufgabe 9a: Erweitert die Datei familie.pl um die Regeln für die Prädikate kind_von/2, mutter/2, und vater/2 und überprüft diese anschließend.

Aufgabe 9b: Erweitert die Datei familie.pl um die Regeln für die Prädikate schwester/2, bruder/2, cousin/2, cousine/2, tante/2, onkel/2, schwager/2, schwaegerin/2, großvater/2 und großmutter/2 und überprüft diese anschließend.

Aufgabe 10: Legt eine Datei namens glueck.pl an und übersetzt die folgende Information in Prolog: *Ursula ist hübsch. Norbert ist reich und gutaussehend. Karla ist reich und stark. Pierre ist stark und gutaussehend. Bruno ist gütig und stark. Alle Männer lieben hübsche Frauen. Alle reichen Männer sind glücklich. Jeder Mann, der eine Frau liebt, die ihn liebt, ist glücklich. Jede Frau, die einen Mann liebt, der sie liebt, ist glücklich. Karla liebt jeden Mann, der sie liebt. Ursula liebt jeden Mann, der sie liebt, vorausgesetzt, daß er entweder reich und gütig oder gutaussehend und stark ist.*

(a) Zeigt mit Prolog, wer hier glücklich ist.
(b) Fügt eine plausible Regel hinzu, die alle glücklich macht.
(c) Sucht in der neuen Version alle Fälle von gegenseitiger Zuneigung, einseitiger Zuneigung und von Rivalitäten (zwei oder mehr Personen lieben die gleich Person).

Kapitel 3.
Semantische Netze

3.1. Einführung

In diesem Kapitel geht es um ein Programm, in welchem die im semantischen Netz aus Abbildung 3.1. kodierte Information enthalten ist. Dieses Programm hat einen konkreten computerlinguistischen Hintergrund, insofern ein semantisches Netz eine mögliche Art darstellt, semantisches bzw. allgemeines Wissen darzustellen, ohne welches die Analyse und Synthese von natürlicher Sprache problematisch ist. Über die Implementierung eines solchen Netzes wird außerdem ein ganz zentrales Konstrukt der Prolog-Programmierung vorgestellt, nämlich die Rekursion.

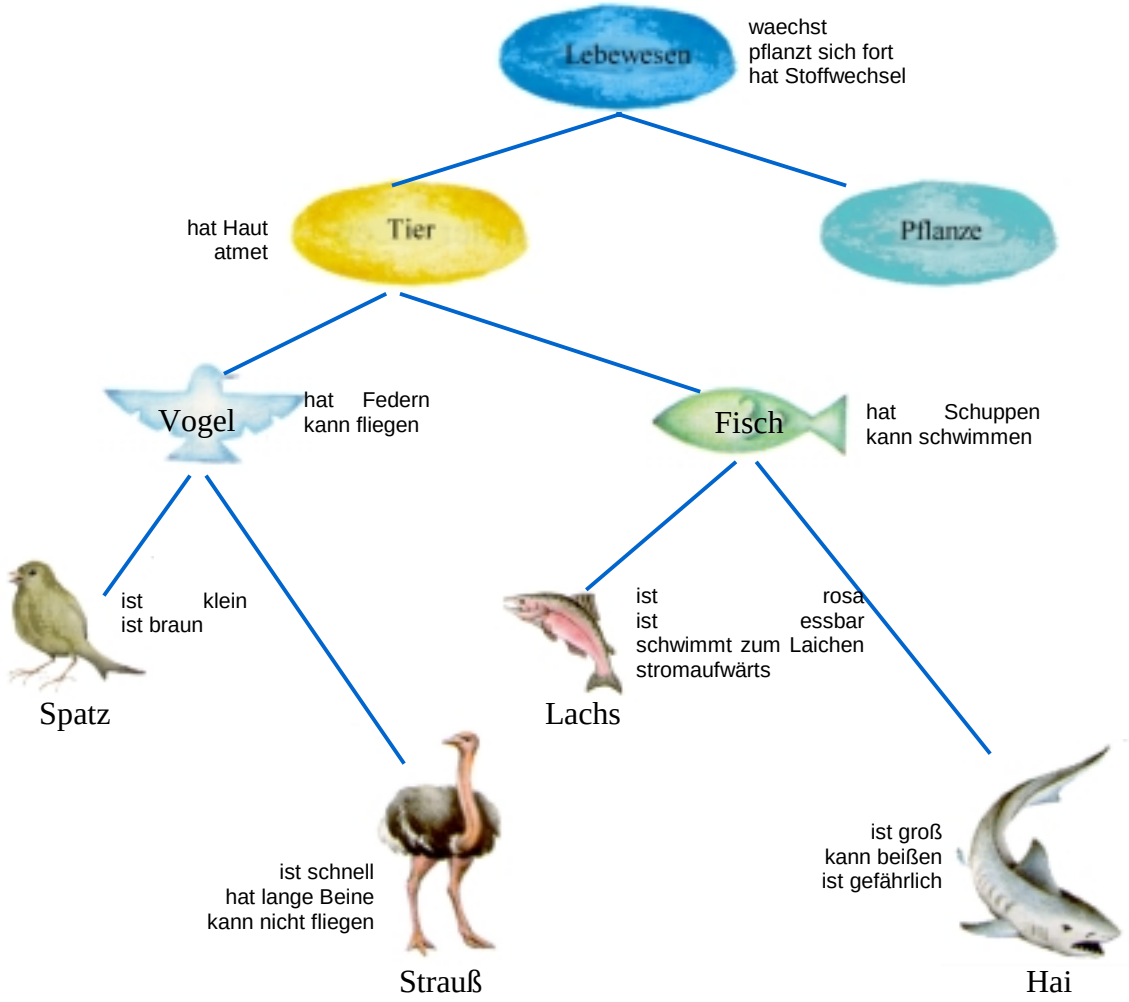


Abb. 3.1. Ausschnitt aus einem semantischen Netz.
(Abbildung nach George Miller, *Wörter. Streifzüge durch die Psycholinguistik*, Spektrum Verlag 1993)

Gegenstand eines semantischen Netzes ist die begriffliche Einordnung von Konzepten sowie die Zuordnung von bestimmten Eigenschaften, die diese Konzepte aufweisen. Ein semantisches Netz weist eine hierarchische Struktur auf und basiert letztendlich auf der aristotelischen Begriffsdefinition, nach der ein beliebiger Begriff definiert ist durch die Angabe seines Oberbegriffes (*Genus Proximum*) und derjenigen Merkmale, die notwendig und hinreichend sind, um den Begriff von anderen Unterbegriffen desselben Oberbegriffes abzugrenzen (*Differentia Specifica*). Diesbezüglich geht ein semantisches Netz allerdings etwas weiter, insofern auch solche Merkmale auftreten können, die eher den Status von zusätzlicher Information haben. Ein wichtiger Punkt (auch für unser Prolog-Programm) ist bei einem semantischen Netz die Tatsache, daß sich die Eigenschaften eine Oberbegriffes auf alle seine Unterbegriffe 'vererben'.

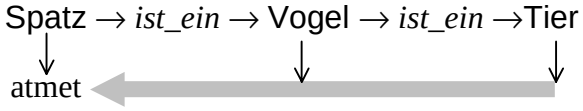
Im Rahmen der KI-Forschung sind semantische Netze spätestens seit den Arbeiten von M.R. Quillian Ende der sechziger Jahre ein Begriff.[11] Quillians zentrales Anliegen war es, ein Modell des menschlichen Gedächtnisses zu entwickeln. In Zusammenarbeit mit A. M Collins wurde eine Version dieses Modells implementiert, und zwar unter der Bezeichnung TLC – '*teachable language comprehender*' – einer der ersten Versuche, menschliches Wissen bzw. die Verarbeitung dieses Wissens auf einem Computer zu modellieren. Wenn auch TLC zahlreiche Mängel hatte, wird es doch allgemein als Vorläufer späterer Implementierungen semantischer Netze betrachtet. Interessant war an TLC, daß es in bestimmten Punkten – z.B. bezüglich der Performanz bei der Verarbeitung von Fragen – Analogien mit der menschlichen Sprachverarbeitung aufwies. Es wurden unter anderem Testreihen aufgestellt, in denen die Probanden durch die Antworten *ja/nein* Aussagen wie in etwa die folgenden verifizieren sollten: *ein Spatz ist ein Tier / ein Hai ist ein Lebewesen / ein Spatz hat Flügel / ein Hai ist gefährlich* usw. Bezüglich der Beantwortungszeit dieser Fragen gab es eine deutliche Analogie zwischen den menschlichen Probanden und TLC insofern die Dauer der Beantwortung mit jeder höheren Netz-Ebene anstieg. Die Bejahung der Aussage *ein Spatz ist braun* kommt schneller als die der Aussage *ein Spatz atmet.* Während spätere Untersuchungen zeigten, daß diese Tatsache nicht zwingend mit der hierarchischen Organisation des semantischen Netzes zu tun haben muß, riefen diese Ergebniss doch zur damaligen Zeit viel Interesse hervor.

Wenn wir Abb. 3.1. betrachten, sehen wir, daß es in diesem Netz prinzipiell jeweils um Fakten über Klassen von Objekten geht, die Unterklassen einer allgemeineren Klasse sind und selber spezifischere Unterklassen haben. Die Klasseninklusion wird durch die Kanten zwischen den einzelnen Begriffen ausgedrückt, die der 'ist_ein' Beziehung entsprechen; linear könnte das so aussehen:

Spatz → *ist_ein* → Vogel → *ist_ein* → Tier.

Ein wichtiger Punkt ist der folgende: durch die Inklusion in der Klasse Vogel, die ihrerseits in der Klasse Tier inkludiert ist, ist auch die Klasse Spatz in der Klasse Tier inkludiert – allerdings nicht direkt, sondern indirekt.

Eine andere Art von Aussage bezieht sich auf die jeweiligen Eigenschaften der Begriffe, wobei hier das Vererbungsprinzip eine zentrale Rolle spielt: die Merkmale eines bestimmten Begriffes 'vererben' sich auf seine Unterbegriffe. Eine bestimmte Klasse weist also nicht nur die Eigenschaften auf, die ihr direkt zugewiesen sind, sondern auch die aller Klassen, von denen es eine Unterklasse ist:



Das Vererbungsprinzip ermöglicht es, die Information auf besonders ökonomische Art und Weise zu erfassen, insofern Redundanz, in diesem Fall das mehrfache Aufführen von Eigenschaften, entfällt. Problematisch wird dieser Punkt natürlich dann, wenn ein Merkmalskonflikt vorliegt – wie z.B. im Falle des Straußes, der nicht fliegen kann. Dazu mehr weiter unten. Was unser Prolog-Programm betrifft, so ist die Formulierung des Vererbungsprinzipes bzw. der indirekten Ober/Unterbegriffsbeziehung der Knackpunkt des Ganzen.

3.2. Die Implementierung des semantischen Netzes in Prolog

Bevor wir unser Prolog-Programm formulieren, machen wir uns zunächst nochmal die Problemstellung genau klar:

1. Die einzelnen Begriffe im Netz entsprechen Klassen von Objekten. Diese sind hierarchisch geordnet, insofern zwischen ihnen jeweils die *ist_ein* Beziehung herrscht.
2. Den Begriffen sind bestimmte Merkmale jeweils direkt zugeordnet
3. Er herrscht das Vererbungsprinzip: die Eigenschaften eines Begriffes werden auf seine Unterbegriffe vererbt — sowohl auf die direkten als auch die indirekten.

[11] Obwohl es auch Vorläufer der semantischen Netze gab, z.B. können dazu die sogenannten *Existential Graphs* von Charles S. Peirce am Ende des 19. Jahrhunderts gezählt werden, die letzendlich die logische Basis für die Konzeptgraphen darstellen (vgl. GKI Skript, Kapitel 4)

Wie im Falle des Familienstammbaumes auch, geht es zunächst darum, die im semantischen Netz enthaltenen Fakten als Prolog-Aussagen zu erfassen. Was den ersten Punkt angeht, so ist diese Aufgabe recht einfach: wir verwenden ein zweistelliges Prädikat ist_ein/2, dessen erstes Argument der Unterbegriff, das zweite Argument dessen direkter Oberbegriff ist:
ist_ein(*Unterbegriff*,*Oberbegriff*); konkret umgesetzt also:

```
ist_ein(tier,lebewesen).
ist_ein(vogel,tier).
ist_ein(fisch,tier).
ist_ein(spatz,vogel).
```

usw.[12]

In einem zweiten Schritt können nun die Eigenschaften, die den Begriffen direkt zugeordnet sind, erfaßt werden. Dazu bietet sich eine Funktor-Argument-Struktur mit dem Funktor merkmal an, wiederum zweistellig: merkmal(*Begriff*,*Merkmal*):

```
merkmal(tier,atmet).
merkmal(tier,waechst).
merkmal(vogel,hat_federn).
merkmal(fisch,hat_flossen).
```

usw.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 1 und 2** am Kapitelende angehen

Das Prolog-Programm ist nun in der Lage, Fragen wie die folgende zu beantworten:

```
?- ist_ein(fisch,X).
X = tier
```

oder

```
?- merkmal(lachs,X).
X = ist_rosa
```

usw.

Das Problem dabei ist aber, daß jeweils nur die unmittelbaren Ober/Unterbegriffspaare, die ja durch das ist_ein/2 Prädikat definiert sind, angezeigt werden. Entsprechend findet die Merkmalszuordnung auch nur auf der unmittelbaren Ebene statt.

Was wir aber eigentlich ausdrücken wollen, ist ja das folgende: Die Eigenschaften einer beliebigen Klasse werden auf alle – also auch die indirekten – Unterklassen übertragen. Wie kann man das in den Griff kriegen? (Das Problem mit dem Merkmalskonflikt (Vogel fliegt, Strauß aber fliegt nicht) wollen wir zunächst unberücksichtigt lassen.)

Nun, wir gehen die Sache, analog zum Familienstammbaum aus Kapitel 3, zunächst einmal informell an und treffen die folgende Aussage:

Ein beliebiger Begriff B hat die Eigenschaft E wenn eine der folgenden Bedingungen gilt:

1. E ist dem Begriff B als Merkmal direkt zugewiesen
2. B ist ein Unterbegriff eines anderen Begriffes B1, und das Merkmal E ist B1 zugewiesen.

Was die erste Bedingung angeht, so ist sie einfach in einer Regel auszudrücken:

```
eigenschaft(B,E):–
        merkmal(B,E).
```

[12] Wer im 2. Kapitel die ' Faustregeln zur Übersetzung von natürlichsprachigen Ausdrücken in Prolog' aufmerksam gelesen hat (was ja wohl hoffentlich auf alle zutrifft!), wird sich vielleicht wundern, daß die Klassenbezeichnungen hier nicht als Prädikate verwendet werden: vogel(X) oder tier(X) usw. Wir behandeln hier Klassen wie Individuen, damit wir Beziehungen zwischen den Klassen (z.B. die Unterbegriffs-Beziehung) direkter formulieren können.

Die Formulierung der zweiten Bedingung ist aber problematisch – die Unterbegriffs-Beziehung, die sowohl die direkten als auch die indirekten Unterbegriffe erfaßt, ist Prolog ja noch nicht bekannt, wir müssen sie erst definieren.

Dieses ist der zentrale Punkt des Programms, über den wir auch eine ganz wesentliche Charaktereigenschaft der Prolog-Programmierung exemplifizieren wollen, nämlich die Formulierung rekursiver Prädikate.

Es geht also darum, ein Prädikat unterbegriff/2 zu definieren, dessen erstes Argument ein Begriff ist, und dessen zweites Argument all die Begriffe sind, von denen das erste Argument Unterbegriff ist – sowohl direkt als auch indirekt. Es soll mithin auf ein Anfrage wie

```
?- unterbegriff(spatz,X).
```

als mögliche Antworten sowohl die direkten als auch die indirekten Oberbegriffe liefern:

```
X = vogel
X = tier
```

Erneut wollen wir die Problemstellung umgangssprachlich formulieren, um diese Aussage dann in Prolog zu übersetzen. Der einfache Fall ist ja der, bei dem es um die direkte Unterbegriffsbeziehung geht. Dieser Fall ist leicht zu erfassen:

1. Ein beliebiger Begriff *B* ist Unterbegriff eines Ausdruckes *B1*, wenn *B* direkter Unterbegriff von *B1* ist.

Wir behalten dabei im Kopf, daß die direkte Unterbegriffs-Beziehung durch die ist_ein/2 Fakten in der Wissensbasis enthalten sind und übersetzen diese Regel wie folgt in Prolog:

```
unterbegriff(B,B1):-
        ist_ein(B,B1).
```

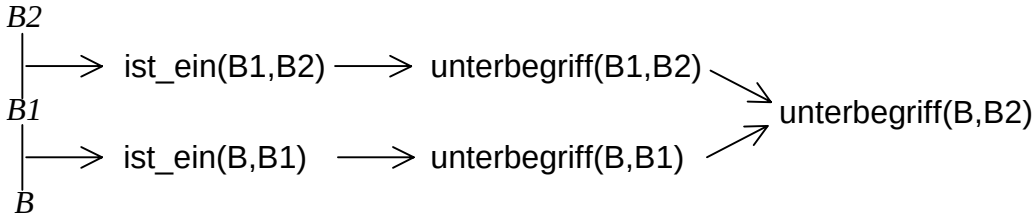
Damit erhalten wir aber eben nur die unmittelbaren Unterbegriffe. Betrachten wir nun eine Möglichkeit, sozusagen eine Stufe höher zu gehen:

2. Ein beliebiger Begriff *B* ist Unterbegriff eines Begriffes *B2*, wenn *B* ein direkter Unterbegriff eines *B1* ist und *B1* ein direkter Unterbegriff von *B2*.

Bei dieser Formulierung geht es nicht mehr um zwei Begriffe, sondern um drei: B, B1 und B2. Begriff B1 ist sozusagen Mittler zwischen B und B2; die Unterbegriffsbeziehung wird hier also auf zwei Ebenen gebracht. In Prolog hätte das diese Form:

```
unterbegriff(B,B2):-
        ist_ein(B,B1),
        ist_ein(B1,B2).
```

Man könnte sich das wie folgt linearisiert verdeutlichen:



Damit wäre die Unterbegriffsbeziehung auf eine weitere Ebene gebracht – aber, und das ist der Haken an der Sache, damit wäre Schluß. Um noch höher in der Netzhierarchie aufzusteigen, müßte man einen weiteren Begriff hinzufügen, in etwa so:

```
unterbegriff(B,B3):-
        ist_ein(B,B1),
        ist_ein(B1,B2)
        ist_ein(B2,B3).
```

Das kann so nicht funktionieren – auf diese Art müßten ja sämtliche Ebenen ausbuchstabiert werden, und das ist nicht Sinn der Sache. Wir verwerfen also diese Möglichkeit und versuchen es auf eine andere Art:

2. Ein beliebiger Begriff *B* ist Unterbegriff eines Begriffes *B2*, wenn *B* ein direkter Unterbegriff eines *B1* ist und *B1* ein **Unterbegriff** von *B2*.

Hier wird nicht, wie weiter oben, auf die **direkte** Unterbegriffsbeziehung zwischen B1 und B2 Bezug genommen, sondern auf die **allgemeine** Unterbegriffsbeziehung – die natürlich die direkte umfaßt. In Prolog sieht das so aus:

```
unterbegriff(B,B1):-
        ist_ein(B,B2),
        unterbegriff(B2,B1).
```

Kriegsentscheidend in dieser Regel ist die Tatsache, daß im Regelrumpf genau die Funktor-Argument-Struktur auftritt (natürlich mit anderen Argumenten), die es durch die Regel erst zu definieren gilt. In diesem Fall ist die Frage erlaubt, wie das überhaupt funktionieren kann – wir wollen das Prädikat unterbegriff/2 definieren, verwenden es aber bereits bei der Definition. Diese Art von Regel wird als REKURSIVE REGEL bezeichnet.

Um dieses Kernprinzip von Prolog zu verstehen, geht es in den nachfolgenden Abschnitten erstmal um Rekursion und die Verarbeitung rekursiver Prädikate von Prolog.

3.2.1. REKURSION

Rekursion als Programmiertechnik ist in der Künstlichen Intelligenz und in bestimmten Programmiersprachen wie z.B. Lisp oder eben Prolog weit verbreitet. Im Grunde genommen ist eine rekursive Regel eine Regel, die auf sich selbst angewendet werden kann – eine rekursive Relation ist eine Relation, die durch sich selbst definiert ist. Das klingt kompliziert, ist aber im Grunde einfach nachzuvollziehen. Nehmen wir dazu ein informelles Beispiel, in dem es ganz natürlich ist, Rekursion zu verwenden, wie z.B. die Aussage

1. *Eine Person X ist ein Römer wenn ihr Vater ein Römer ist.*

Eine weitere Angabe könnte lauten:

2. *Eine Person X ist Römer, wenn sie in Rom geboren wurde*.

Durch die Definitionen 1. (rekursiv) und 2. ist nun die 'Bürgerzugehörigkeit' leicht nachweisbar: Person X ist entweder Römer, weil sie in Rom geboren ist, oder weil ihr Vater Römer ist. Der Vater der Person X kann entweder Römer sein, weil er in Rom geboren wurde, oder weil sein Vater Römer war. Dieser kann entweder Römer sein, weil er Rom geboren wurde – und so weiter und so fort.

Das Prinzip bei der Rekursion ist daher, eine einfache 'Erfolgsbedingung' zu formulieren (in dem Römer-Beispiel also die Bedingung 2.), und die rekursive Bedingung auf eine Art auszudrücken, daß sie quasi solange wiederholt wird, bis diese einfache Erfolgsbedingung erreicht wird.

Genau diesen Anforderungen entspricht das Prädikat unterbegriff/2. Wie aber wird ein solches Prädikat bei konkreten Anfragen von Prolog verarbeitet? Führen wir uns, um diesen Punkt zu klären, zunächst einen Ausschnitt aus der Wissensbasis vor Augen:[13]

Fakten	Regeln
1. ist_ein(spatz,vogel). 2. ist_ein(strauss,vogel). 3. ist_ein(vogel,tier). 4. ist_ein(hai,fisch). 5. ist_ein(lachs,fisch). 6. ist_ein(fisch,tier). 7. ist_ein(tier,lebewesen).	A. unterbegriff(B,B1):- ist_ein(B,B1). B. unterbegriff(B,B1):- ist_ein(B,B2), unterbegriff(B2,B1).

In der linken Spalte sind die ist_ein/2 Fakten eingegeben; in der rechten Spalte stehen zwei Regeln, die bestimmen, unter welchen Bedingungen für ein B und B1 gilt, daß zwischen ihnen die Unterbegriffs-

[13] Die Fakten und Regeln sind in dieser Darstellung durchnumeriert, damit wir uns besser darauf beziehen können.

Beziehung herrscht. Entweder ist B ein direkter Unterbegriff von B1 (der einfache Fall, die Endbedingung, Regel A.), oder es ist direkter Unterbegriff eines B2, welches seinerseits Unterbegriff von B1 ist (rekursive Definition, Regel B.). Was die Regeln betrifft, so geben die Aussagen auf der rechten Seite der Regel (also die Aussagen im Regelrumpf) diejenigen Bedingungen wieder, die erfüllt werden müssen, damit die Regel erfolgreich ist. Zur besseren Lesbarkeit wollen wir diesen Sachverhalt nochmal wie folgt darstellen:

Regel A):

Regelkopf	Regelrumpf
Diese Aussage ist bewiesen, wenn	*diese Aussage bewiesen ist:*
unterbegriff(B,B1):-	**ist_ein(B,B1).**

Regel B):

Regelkopf	Regelrumpf	
Diese Aussage ist bewiesen, wenn	*diese Aussage bewiesen ist und*	*diese Aussage bewiesen ist*
unterbegriff(B,B1):-	**ist_ein(B,B2),**	**unterbegriff(B2,B1).**

Was passiert, wenn Prolog auf der Grundlage dieser Fakten und Regel Anfragen wie die folgenden beantworten soll:

1) ?- unterbegriff(hai,fisch).
2) ?- unterbegriff(spatz,tier).

Beginnen wir mit der Anfrage 1), in der bewiesen werden soll, daß 'Spatz' Unterbegriff von 'Vogel' ist. Prolog ruft Regel A) auf, und zwar mit entsprechend instantiierten Argumenten,[14] also

```
unterbegriff(spatz,vogel):-
        ist_ein(spatz,vogel).
```

Damit die Anfrage erfolgreich ist, muß die Bedingung, die im Regelrumpf formuliert ist, erfüllt sein. In diesem Fall reicht es aus, die Wissensbasis nach einem entsprechenden Fakt zu durchsuchen. Da dieser Fakt gefunden werden kann (Nr. 4 in der Tabelle), ist der Beweis für die Anfrage erbracht, die Antwort von Prolog lautet Yes.

Kommen wir zu Anfrage Nr. 2. Wird Regel A) mit entsprechend instantiierten Argumenten aufgerufen:

```
unterbegriff(spatz,tier):-
        ist_ein(spatz,tier).
```

führt dieses nicht zum Erfolg – schließlich gibt es kein entsprechendes Fakt in Wissensbasis. In einem solchen Fall wird die nächste Regel aufgerufen, hier also Regel B):

```
unterbegriff(spatz,tier):-
        ist_ein(spatz,B2),
        unterbegriff(B2,tier).
```

Der Regelrumpf umfaßt hier zwei Bedingungen, die sogenannten Teilziele, die beide erfüllt werden müssen, damit die Anfrage bewiesen werden kann. Prolog beginnt mit dem ersten Teilziel: ist_ein(spatz,B2). Dieses Teilziel kann durch Prüfung der Wissensbasis bewiesen werden, indem B2 an den Wert vogel gebunden wird (Fakt Nr. 1). Prolog ersetzt nun die Variable X durch vogel:

[14] Das bedeutet, daß statt einer Variablen der konkrete Wert aus der Anfrage eingesetzt wird, und zwar sowohl im Regelkopf als auch in den Bedingungen des Regelrumpfes.

```
unterbegriff(spatz,tier):-
        ist_ein(spatz,vogel),      → Fakt in der Wissensbasis
        unterbegriff(vogel,tier).  → muß als nächstes bewiesen werden
```

Das nächste Teilziel lautet also unterbegriff(vogel,tier). An dieser Stelle wird das gesamte Prädikat neu aufgerufen – diesmal mit demArgumenten 'vogel' und 'tier'. Dies ist der entscheidende Punkt, nun wird klar, was es bedeutet, daß eine Regel 'auf sich selbst angewendet werden kann'. Da für vogel gilt, daß es in direkter Unterbegriffsbeziehung zu tier steht, führt bereits die 'Endbedingung' zum Erfolg: Die Bedingung im Regelrumpf von Regel A), hier mit entsprechend instantiierten Argumenten:

```
unterbegriff(vogel,tier):-
        ist_ein(vogel,tier).
```

steht als Fakt in der Wissenbasis (Nr. 3 in der Tabelle), also ist nachgewiesen, daß vogel ein Unterbegriff von tier ist, und also spatz ebenfalls, die Antwort lautet Yes.

Diesen an sich recht einfachen Vorgang schrittweise schriftlich zu illustrieren, ist sehr papieraufwendig und täuscht über die eigentliche Geschmeidigkeit dieser Programmiertechnik hinweg. Deshalb wird an dieser Stelle nicht näher auf diesen Punkt eingegangen, stattdessen wird einerseits auf die animierte Demonstration in der Veranstaltung, andererseits auf das Kapitel über Unifikation verwiesen, in welchem die Verarbeitungsweise rekursiver Prädikate erneut aufgegriffen wird.

Da wir nun über das Prädikat unterbegriff/2 verfügen, ist es jetzt auch einfach, die Eigenschafts-Zuweisung entsprechend zu formulieren. Wir rekapitulieren:

Ein beliebiger Begriff B hat die Eigenschaft E wenn eine der folgenden Bedingungen gilt:

1. E ist dem Begriff B als Merkmal direkt zugewiesen
2. B ist ein Unterbegriff eines anderen Begriffes B1, und das Merkmal E ist B1 zugewiesen.

Die erste Bedingung lautete:

```
eigenschaft(B,E):-
        merkmal(B,E).
```

Die zweite Bedingung ist rekursiv und lautet wie folgt:

```
eigenschaft(B,E):-
        unterbegriff(B,B1), eigenschaft(B1,E).
```

Damit hat das Programm die folgende Form:

```
ist_ein(spatz,vogel).
ist_ein(strauss,vogel).
ist_ein(vogel,tier).
ist_ein(hai,fisch).
ist_ein(lachs,fisch).
ist_ein(fisch,tier).
ist_ein(tier,lebewesen).
ist_ein(pflanze,lebewesen).

merkmal(lebewesen,waechst).
merkmal(tier,atmet).
merkmal(tier,hat_haut).
merkmal(vogel,kann_fliegen).
merkmal(vogel,hat_federn).
merkmal(fisch,kann_schwimmen).
merkmal(fisch,hat_schuppen).
merkmal(spatz,ist_braun).
merkmal(strauss,ist_schnell).
merkmal(strauss,kann_nicht_fliegen).
merkmal(hai,ist_gefaehrlich).
merkmal(lachs,ist_rosa).

eigenschaft(B,E):-
        merkmal(B,E).
eigenschaft(B,E):-
        unterbegriff(B,B1),
        eigenschaft(B1,E).

unterbegriff(B,B1):-
        ist_ein(B,B1).

unterbegriff(B,B1):-
        ist_ein(B,B2),
        unterbegriff(B2,B1).
```

Das semantische Netz als Prolog-Programm

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgabe3** am Kapitelende angehen

Mit den rekursiven Definitionen von unterbegriff/2 und eigenschaft/2 ist gewährleistet, daß auf eine Anfrage wie z.B. ?- merkmal(spatz,X) die folgenden Antworten erscheinen: X = ist_braun ;X = kann_fliegen ;X = hat_federn ;X = atmet ;X = hat_haut ;X = waechst; usw.

Damit hätte das Programm 'semantisches Netz' seine endgültige Form, wenn da nicht ein paar kleine Schönheitsfehler wären - z.B. der Merkmalskonflikt bei Strauß. Außerdem ist der Umgang mit dem Programm nicht gerade benutzerfreundlich – weder was die Anfragen noch die was Ausgabe der Antworten betrifft (die vollständig nur durch wiederholte Betätigung des Semikolons ausgegeben werden).

Der Revidierung dieser Mängel sind die nachfolgenden Abschnitte gewidmet, in denen außerdem ein wichtiger Aspekt der Prolog-Programmierung eingeführt wird: bei der Formulierung des Stammbaum-Programmes und der Implementierung des semantischen Netzes sind wir sozusagen 'from scratch' ausgegangen – alle Prädikate sind rein selbstdefiniert. Es gibt aber zahlreiche vordefinierte Prolog-Prädikate, deren Verwendung dem Programmierer das Leben erleichtern und ohne die er in vielen Fällen gar nicht weiterkommt. Einige dieser vordefinierten Prädikate werden wir zur Verbesserung des semantischen Netzes verwenden.

3.3. Die Modifikation des Semantischen Netzes I: der Merkmalskonflikt

Das konkrete Problem besteht darin, daß der Strauß von seinem Oberbegriff Vogel die Eigenschaft erbt, fliegen zu können – ihm selber ist aber das Merkmal 'kann nicht fliegen' zugewiesen. Verallgemeinert also läßt sich die Problemstellung wie folgt als Frage formulieren: wie handelt man, wenn ein Unterbegriff ein Merkmal aufweist, welches im direkten Konflikt mit einem von einem Oberbegriff vererbten Merkmal steht? (Diese Frage taucht übrigens in der einen oder anderen Form immer wieder im Zusammenhang mit semantischen Netzen oder ähnlichen Konstrukten, wie z.B. Typhierarchien[15] auf).

Was die Bewältigung dieses Problemes in Prolog angeht, so gibt es durchaus Möglichkeiten, die Sache systematisch in den Griff zu bekommen. Im vorliegenden Skript aber verwenden wir eine Lösung, die eher einen ad-hoc Charakter hat – hauptsächlich deshalb, weil diese Lösung nur eine geringe Modifikation des Programmes beinhaltet. Zunächst aber erfolgen doch eine paar Anmerkungen, die sich auf eine systematische Regelung beziehen.

Das dem Strauß direkt zugewiesen Merkmal kann_nicht_fliegen ist die Negation des dem Vogel direkt zugewiesenen Merkmales kann_fliegen. Die Art und Weise, in der wir im Programm diese Fakten eingegeben haben, erlaubt es Prolog aber nicht, diesen Zusammenhang zu erkennen – kann_nicht_fliegen und kann_fliegen sind jeweils Atome, deren eigentliche Interpretation nur durch einen Menschen erfolgen kann. Das bedeutet, daß Prolog bei dieser Datenlage nicht fähig ist, den systematischen Zusammenhang zwischen diesen Merkmalen zu erkennen. Das bedeutet weiterhin, daß eine Regel, die sich darauf bezieht, daß bei einem solchen Merkmalskonkflikt immer dasjenige Merkmal Vorrang hat, welches einem Element zugeordnet ist, daß in der Hierarchie weiter unten steht (und genau das ist der entscheidende Punkt!) hier nicht formuliert werden kann. Anders ausgedrückt: hat ein Element ein beliebiges Merkmal [+merkmal] und dessen Oberbegriff hat das Merkmal [–merkmal], so hat immer das dem Element direkt zugewiesene Merkmal Vorrang. Auf den konkreten Fall bezogen: das Merkmal kann_nicht_fliegen hat mit Bezug auf den Strauß Vorrang vor dem Merkmal kann_fliegen. Das Problem ist aber: der Zusammenhang zwischen den Merkmalen kann_fliegen bzw. kann_nicht_fliegen, der für den Programmierer klar gegeben ist, ist für Prolog so nicht erkennbar.

Um also diese Problemstellung systematisch zu implementieren, wäre es einerseits erforderlich, die Fakten über die Merkmale neu zu definieren, und zwar so, daß bestimmte Beziehungen (z.B. die Negation) zwischen den Merkmalen für Prolog transparent werden, andererseits müßte die Regel über die Zuweisung der Eigenschaften umformuliert werden und dem Fall des Merkmalskonfliktes so Rechnung tragen, daß stets das direkt zugewiesene Merkmal das ererbte Merkmal außer Kraft setzt.

Kommen wir nun jedoch zu unserer ad-hoc Lösung. Diese besteht zunächst in der Hinzufügung von Fakten, die sich auf Merkmalskonflikte beziehen. Wir definieren ein einfaches Prädikat konflikt/2, dessen

[15] siehe dazu Skript GK I, Kapitel 4

erstes Argument ein Begriff ist und dessen zweites Argument dasjenige Merkmal, welches der Begriff erbt, aber nicht aufweisen soll. Beispiel: konflikt(strauss,kann_fliegen).[16] Hier wird die Beziehung, die zwischen 'kann_fliegen' und 'kann_nicht_fliegen' besteht, quasi ausbuchstabiert – es wird genau das Merkmal benannt, welches der Oberbegriff vererbt.

Dreh- und Angelpunkt dieser Modifikation ist aber die Veränderung von eigenschaft/2. Dieses Prädikat wird mit Bezug auf konflikt/2 umformuliert, wozu wir den vordefinierten Operator not/1 brauchen. Dieses ist ein Operator zur Programmablaufsteuerung, welcher fehlschlägt, wenn sein Argument (ein beliebiger Term) erfolgreich ist und andersherum. Für mehr Information zu not/1 siehe das Kapitel über Syntax.

eigenschaft/2 hatte die folgende Form (rechte Spalte umgangssprachlich):

eigenschaft(B,E):- merkmal(B,E).	Begriff B hat Eigenschaft E falls E dem B direkt zugewiesen ist (*Mermkal-Fakten*)
eigenschaft(B,E):- unterbegriff(B,B1), eigenschaft(B1,E).	Begriff B hat Eigenschaft E falls B ein Unterbegriff von B1 ist und B1 die Eigenschaft E hat.

Was die erste Regel angeht, so kann diese unverändert bleiben – die einem Begriff direkt zugewiesenen Merkmale sind ja sozusagen unantastbar. Die zweite Regel aber muß verändert werden. Wie das aussehen könnte, wird erstmal wieder umgangssprachlich dargestellt:

Begriff B hat Eigenschaft E falls
B ein Unterbegriff von B1 ist und
B1 die Eigenschaft E hat und
kein Merkmalskonflikt in Bezug auf B und E vorliegt.

Das letzte Teilziel klingt fast zu einfach, um wahr zu sein – da wir aber entsprechende Fakten in der Wissensbasis haben (nämlich konflikt/2), können wir es unter Verwendung von not/1 direkt in Prolog übersetzen.

```
eigenschaft(B,E):-
        unterbegriff(B,B1),
        eigenschaft(B1,E),
        not(konflikt(B,E)).
```

Dadurch ist erreicht, daß Prolog bei entsprechenden Anfragen nicht mehr sich widersprechende Merkmale ausgibt. Getrübt ist diese Lösung aber, wie bereits gesagt, dadurch, daß sie nicht von Prolog aus angemessen formulierten Fakten und Regeln selbständig abgeleitet werden kann, sondern über die konflikt/2-Fakten zustande kommt.

3.4. Die Modifikation des Semantischen Netzes II: Benutzerfreundlichkeit

Zum Abschluß dieses Kapitels geht es darum, das Programm so zu modifizieren, daß der Umgang damit einfacher ist. Dieser Punkt ist eine ganz wichtige Aufgabe der Programmierer – in der Regel werden Programme ja für Benutzer gemacht, die von den Programminterna und der Programmiersprache keine Kenntnisse haben, und von denen auch nicht erwartet werden darf, daß sie diese Kenntnisse haben. Ein Beispiel: die mühsame Verwendung des Semikolons zur Ausgabe aller Eigenschaften, die ein Begriff aufweist. Ein weiterer Punkt: die Anfragen müssen in Form einer Prolog–Klausel (hier einer Funktor-Argument-Struktur) formuliert werden. Diese Punkte sollen geändert werden. Wie wollen eine Prozedur schreiben, die folgendes bewirkt:

[16] Vielleicht gehört dazu ja auch so etwas wie konflikt(hai,hat_schuppen).

Die Eingabe des Wortes 'semnetz' auf der Prolog–Oberfläche veranlaßt das Programm zu der folgenden Aussage:

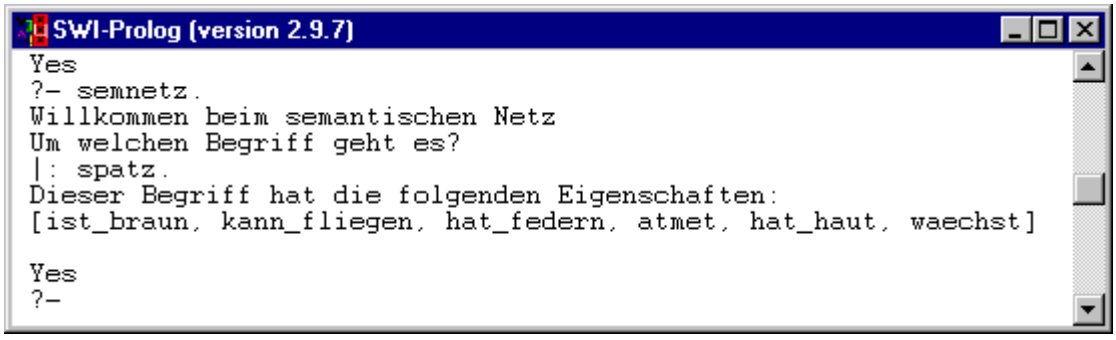
'Willkommen im semantischen Netz. Um welchen Begriff geht es?'

Die Benutzer können daraufhin eine beliebigen Begriff aus dem Netz eingeben, z.B. 'spatz', und erhalten die Ausgabe:

'Dieser Begriff hat die folgenden Eigenschaften:

[waechst, atmet, hat_haut usw]'

Der nachfolgende Screen-Shot zeigt, wie es aussehen soll:

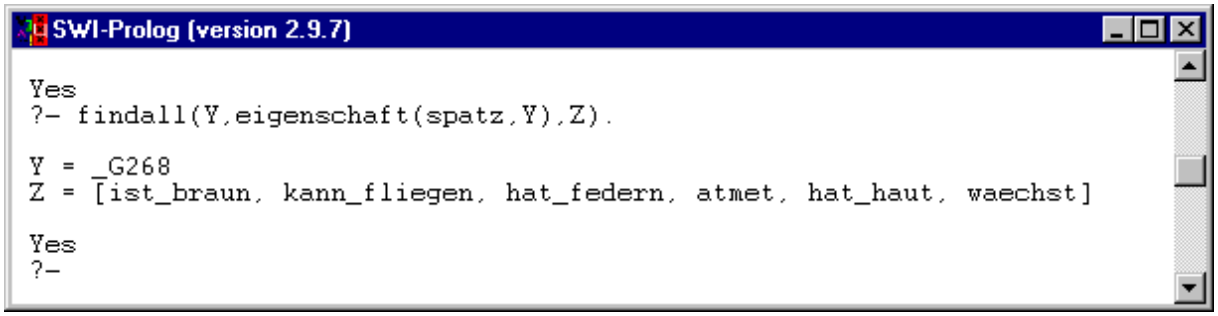


Was wir für diese Prozedur brauchen, sind die vordefinierten Prädikate write/1, read/1, nl, und findall/3.

Beginnen wir mit findall/3. Die Aufgabe dieses treffend benannten Prädikates ist es, für eine beliebige Anfrage alle Lösungsalternativen zu finden und in Form einer Liste (siehe Kapitel über Syntax) auszugeben. Für die Benutzer bedeutet das: mit findall/3 kann die Lösungssuche durch das Semikolon umgangen werden. Um es in unser Programm einzubauen, schreiben wir ein neues Prädikat eigenschaft/1, dessen Argument der Begriff ist, dessen Eigenschaften ermittelt werden soll. Aufgabe dieses Prädikates ist es,

- mit findall/3 alle Eigenschaften des Begriffes zu ermitteln
- mit write/1 diese Eigenschaften auf dem Bildschirm anzuzeigen.

Um den ersten Punkt zu leisten, muß zunächst erklärt werden, wie findall/3 verwendet wird. Das erste Argument von findall/3 ist eine Variable für den zu findenden Wert[17], das zweite Argument ist die Zielklausel, um die es geht, (in unserem Fall: eigenschaft(B,E)), das dritte Argument schließlich ist die Liste der gesammelten Lösungen. Beispielsweise liefert die Anfrage
?- findall(Y,eigenschaft(vogel,Y),Z). eine Antwort wie im nachstehenden Screen-Shot:



Was hier von Interesse ist, ist also die Liste, die sich aus den gesammelten Lösungen für Y ergibt. Kommen wir nun zu unserem einstelligen eigenschaft/1: Dessen Argument war, wie besprochen, ein Begriff. Das erste Teilziel im Regelrumpf ruft dann findall/3 mit entsprechenden Argumenten auf:

```
eigenschaft(B):-
        findall(E,eigenschaft(B,E),L).
```

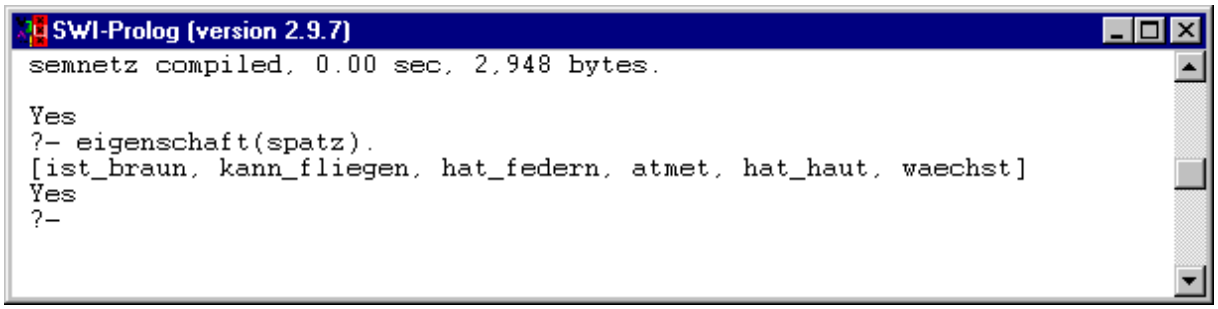
Hier gibt es aber ein Problem: Prolog wird durch nichts veranlaßt, dem Benutzer die Liste L der Eigenschaften auch tatsächlich mitzuteilen. Wird dieses Prädikat nicht umformuliert, so kommen als

[17] Im Grunde kann hier auch ein Term stehen, dessen Argumente mit den in der Zielklausel befindlichen Argumenten unifiziert werden. Da die für das Verständnis dieses Sachverhaltes erforderliche Terminologie noch nicht eingeführt wurde, wird hierauf nicht näher eingegangen.

mögliche Antworten nur Yes oder No. Nun, dieser Zustand kann schnell revidiert werden, dazu dient das eingebaute Prädikat write/1, dessen Argument auf dem Bildschirm ausgegeben wird:

```
eigenschaft(B):-
        findall(E,eigenschaft(B,E),L),
        write(L).
```

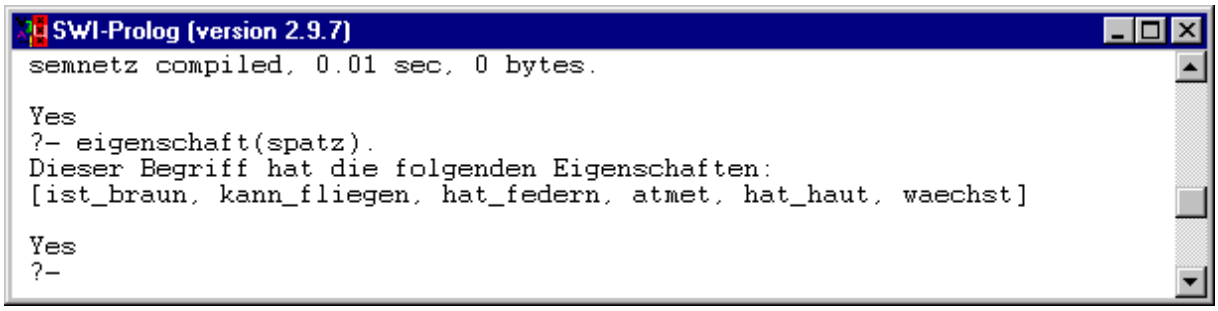
Das Resultat sieht bei entsprechender Anfrage (hier: eigenschaft(spatz)) doch schon viel ansprechender aus als die umständliche alte Version:



Noch freundlicher wird es, wenn dem Benutzer auch noch nett mitgeteilt wird, was ihm ausgegeben wird. Dazu verwenden wir erneut das Prädikat write/1, dessen Argument in diesem Fall der Satz 'Dieser Begriff hat die folgenden Eigenschaften' ist:

```
eigenschaft(B):-
        findall(E,eigenschaft(B,E),L),
        write('Dieser Begriff hat die folgenden Eigenschaften'),
        nl,
        write(L).
```

Wichtig: den umgangssprachlichen Satz unbedingt in einfache Anführungszeichen setzen – sonst funktioniert das Ganze nicht! 'nl' steht schlicht für *new line*, also 'neue Zeile', und bewirkt einen Zeilenwechsel in der Ausgabe, was die Übersichtlichkeit verbessert. Die Auswirkung dieser Modifikation illustriert der nachstehende Screen-Shot:



Damit sind wir aber noch nicht fertig. Das eigentliche Ziel ist es ja, ein nullstelliges Prädikat semnetz/0 zu formulieren, welches die Benutzer erstmal willkommen heißt und ihnen dann die Möglichkeit bietet, den Begriff als Atom einzugeben. Der Dialog zwischen Programm und Benutzer ist sehr einfach zu erstellen, des eigentlich komplizierten Teilziel von semnetz/0 haben wir bereits durch eigenschaft/1 erledigt. Wichtig ist nur, das eben dieses eigenschaft/1 von semnetz/0 aus aufgerufen wird. semnetz/0 könnte die folgende Form haben:

```
semnetz:-
        write('Willkommen im semantischen Netz.'),
        nl,
        write('Um welchen Begriff geht es?'),
        nl,
        read(B),
        eigenschaft(B).
```

Gehen wir die Teilziele dieses Prädikates einmal der Reihe nach durch, neu und wichtig ist besonders Punkt (5):

1. läßt den Satz 'Willkommen im sem. Netz' auf dem Bildschirm erscheinen
2. bewirkt einen Zeilenumbruch
3. läßt die Frage 'Um welchen Begriff geht es' auf dem Bildschirm erscheinen
4. bewirkt einen Zeilenumbruch
5. liest den vom Benutzer eingegebenen Begriff ein. Im konkreten Programm taucht an dieser Stelle ein blinkender Cursor auf dem Bildschirm auf, Prolog wartet also auf die Eingabe eines Benutzers
6. ruft eigenschaft/1 mit eben diesem Begriff auf (sowohl in (e) wie in (f) die Variable B!)

Durch diese Modifikationen hat das Programm die abschließende gewünschte Form.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 4 und 5** am Kapitelende angehen

Übungen zu Kapitel 3

Aufgabe 1: Legt eine Datei semnetz.pl mit den Fakten über die *ist_ein*-Beziehung an.

Aufgabe 2: Erweitert semnetz.pl um die Fakten über die Merkmale der Begriffe.

Aufgabe 3: Erweitert semnetz.pl um die Regeln für Eigenschaft und Unterbegriff und testet das Programm

Aufgabe 4: Behebt das Problem mit dem Merkmalskonflikt, indem Ihr
a) die Wissensbasis um konflikt/2-Fakten anreichert
b) das Prädikat eigenschaft/2 entsprechend umformuliert

Aufgabe 5: Steigert die Benutzerfreundlichkeit, indem Ihr die im Abschnitt 3.4 vorgestellten Modifikationen in der Datei semnetz.pl realisiert.

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
```

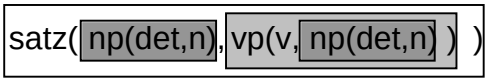
Hier ist satz ein Funktor mit zwei Argumenten, nämlich nominalphrase und verbalphrase, die ihrerseits Atome (Konstanten) sind. Es ist erkennbar, daß die Prädikat-Argument-Strukturen von Fakten Strukturen in diesem allgemeinen Sinne sind.

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, ..., t_n Terme, dann ist $f(t_1,...,t_n)$ 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.

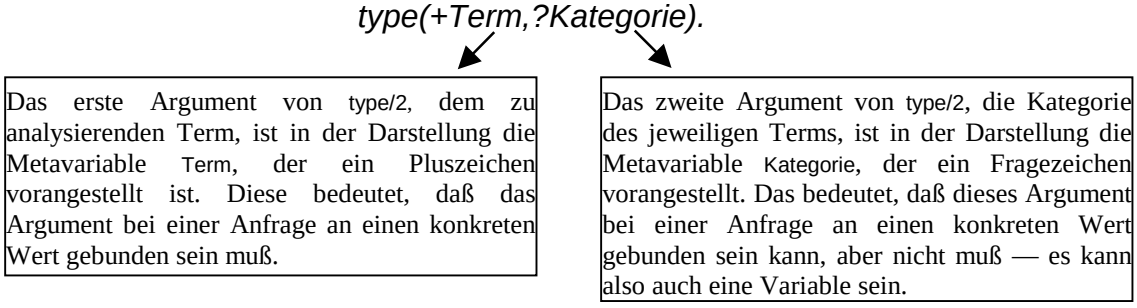
4.3.1. EXKURS: NOTATIONSKONVENTIONEN FÜR PRÄDIKATE

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.

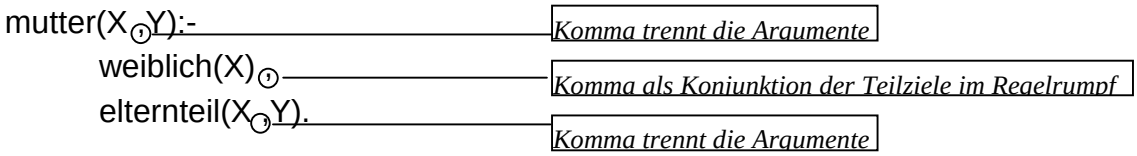
Eine besondere Sorte von Atomen heißt OPERATOREN, die nichts weiter als syntaktisch abweichend auftretende Struktur-Funktoren sind.

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:

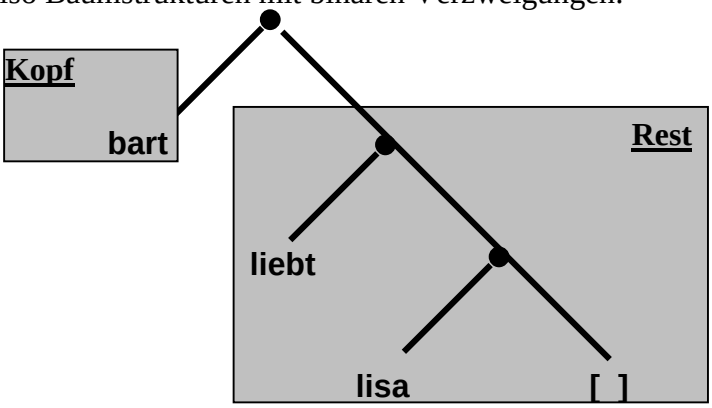


Abb. 4.2.

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:

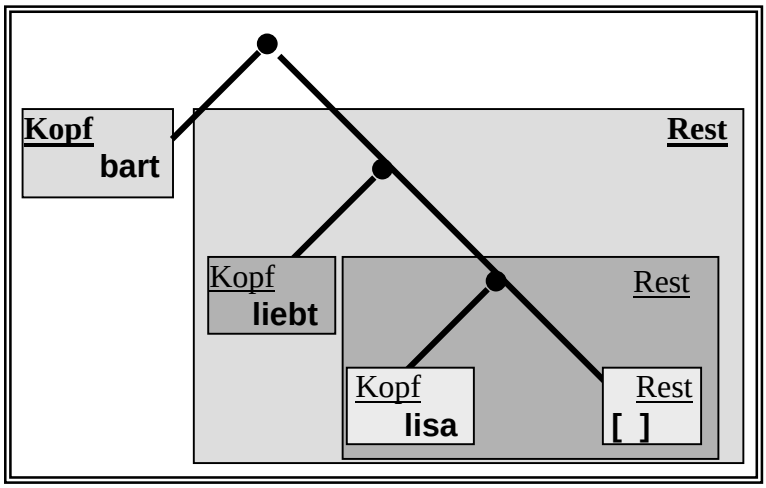


Abb. 4.3.

Ein weiteres Beispiele:

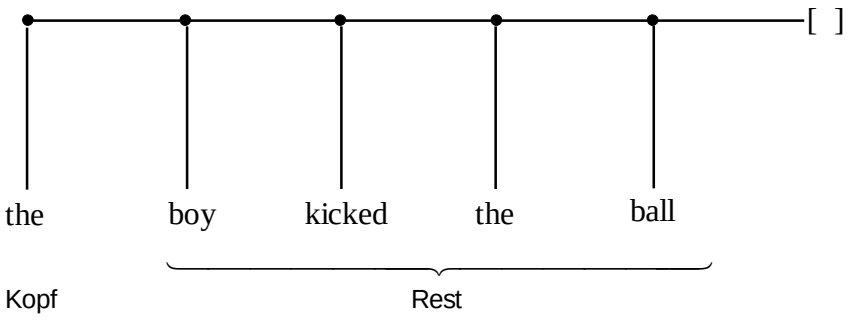


Abb. 4.4. Binnenstruktur von [the, boy, kicked, the, ball]

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 | [ ]].

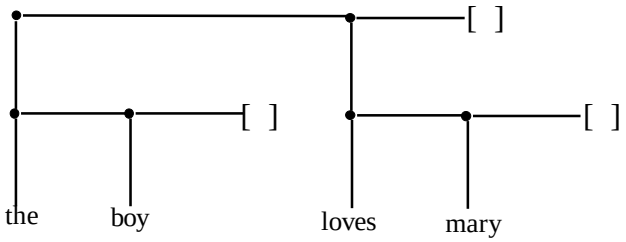


Abb. 4.5. [[the, boy], [loves, mary]]

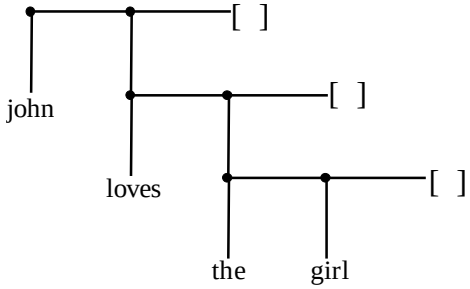


Abb. 4.6. [john, [loves, [the, girl]]

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.

Kapitel 5.
Listenverarbeitung

Die Liste ist ein sehr flexibler Datentyp, der sich besonders auch für linguistische Anwendungen eignet. Allerdings können wir, mit Ausnahme des Kopfes, auf die Elemente einer Liste nicht direkt zugreifen. Zur Verarbeitung von Listen benötigen wir also eine Reihe von Prädikaten, die entweder als System- bzw. Bibliotheksprädikate zur Verfügung gestellt werden, oder die wir auf der Grundlage von elementaren vordefinierten Operationen selbst definieren können. Die Formulierung dieser Prädikate ist eigentlich gar nicht so schwer. Es ist aber wichtig, dafür immer im Hinterkopf zu behalten, wie die interne Struktur einer Liste aussieht, daß es sich bei Listen eben nicht um bloße Aneinanderreihungen von Elementen handelt, sondern um komplexe rekursive Strukturen.

5.1. Listenzerlegung

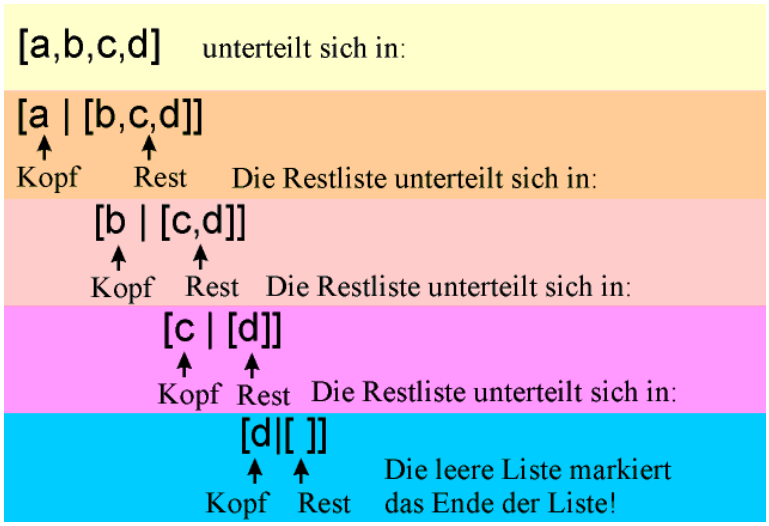
Die wichtigste Grundoperation, die allen anderen Listenoperationen zugrunde liegt, ist die Zerlegung einer Liste in den Listenkopf und den Listenrest. Dies geschieht sehr einfach mithilfe des vordefinierten Listenoperators '|'. Genauer gesagt: ist eine beliebige Liste L gegeben, muß diese Liste mit dem Listenstrukturmuster [Kopf | Rest] in Überbereinstimmung gebracht werden: L = [Kopf | Rest]. Der Knackpunkt dabei ist, wie im letzten Kapitel deutlich wurde, die Tatsache, daß der Rest selber wieder eine Liste ist. Ist L beispielsweise die Liste [john, loves, mary], dann würde, da diese Liste mit Operator als [john | [loves, mary]] wiederzugeben wäre, die Variable Kopf mit john und die Variable Rest mit [loves, mary] unifiziert werden:

```
[Kopf   |      Rest]
  =             =
[john   | [loves, mary]]
```

Auf diesem Hintergrund und unter der Berücksichtigung der Tatsache, daß eine jede Liste stets mit der leeren Liste endet, ist die folgende Entsprechung gut nachzuvollziehen:

[a,b,c,d] entspricht [a | [b | [c | [d | [ ]]]]]

In einzelne Schritte unterteilt, gibt die nachstehende Grafik denselben Sachverhalt wieder:



Wie im vorherigen Kapitel aus der Definition für Listen deutlich wurde, sind diese (wie ja auch Operator-Operand-Strukturen) nur eine besondere Schreibweise der wohlbekannten Funktor-Argument-Strukturen. Der Funktor ist im Falle der Liste ein einfacher Punkt: '.', dahinter stehen, in Klammern, die Argumente dieses Funktors. Intern übersetzt Prolog die Listen auch in Funktor-Argument-Strukturen. Um auf unser Beispiel zurückzukommen, kann also folgende Gleichung genannt werden:

[a,b,c,d] entspricht [a | [b | [c | [d | [ ]]]]] entspricht .(a, .(b, .(c, .(d,[ ]))))

Ein weiteres Beispiel, bei welchem das dritte Element eine eingebettete Liste ist:

[1,2,[a,b],3]
entspricht
[1| [2 | [[a | [b | [ ]]] | [3 | [ ]]]]]
entspricht
.(1, .(2, .(.(a, .(b, [ ])), .(3, [ ]))))

Man sieht, daß die jeweils erste Schreibweise am lesefreundlichsten ist – sie birgt aber möglicherweise die Gefahr, die komplexe rekursive Struktur, die eine Liste aufweist, zu übersehen. Die folgende kleine Aufwärmübung soll einzig das Ziel haben, sich dieser Tatsache bewußt zu werden, indem die einfachen Listenstrukturen jeweils übersetzt werden sollen in Listen mit Listenoperator einerseits und Funktor–Argument–Strukturen andererseits:

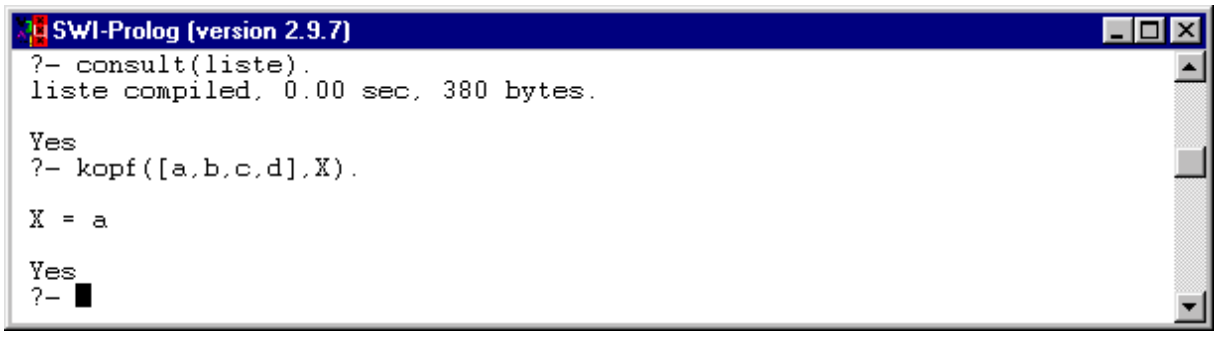
Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgabe 1** am Kapitelende angehen

Soviel zu der internen Repräsentation von Listen. Es ging im wesentlich darum, erneut ein wenig darauf herumzureiten, daß es sich bei Listen eben nicht um eine einfache Aneinanderreihung von Elementen handelt. In den nachfolgenden Abschnitten wird allerdings die einfache Schreibweise verwendet.

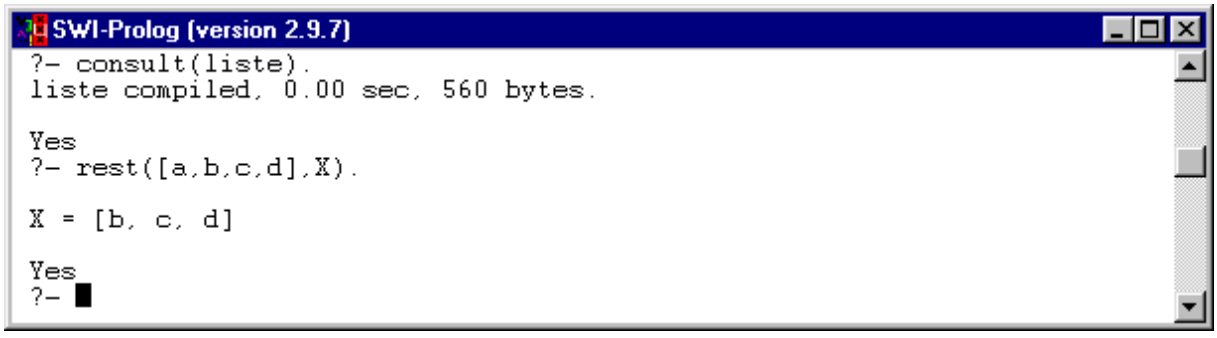
Als nächstes wollen wir eine Reihe von listenverarbeitenden Prädikaten definieren, die jeweils recht unterschiedliche Aufgaben erfüllen. Diese Prädikate werden nicht unbedingt alle später in anderen Programmen eingesetzt – einige dienen nur dazu, einen ersten Einstieg in den Umgang mit Listen zu erhalten und ein Gefühl für die Listenverarbeitung zu gewinnen – beispielsweise über die folgenden Prädikate der folgenden Übung:

> **kopf/2** und **rest/2**, die zu einer beliebigen Liste jeweils den Kopf bzw. den Rest ausgeben. Beispielsweise soll gelten:
> kopf([a, b, c,d], a)
> rest([a, b, c,d], [b,c,d])

Es geht also um die Definition zweier Prädikate, die von einer beliebigen Liste entweder den Kopf oder den Rest ausgeben. Das Prädikat kopf/2 soll mithin die Anfrage ?- kopf([a,b,c,d],X) wie folgt beantworten:



Entsprechend rest/2:



Die Frage ist, wie man ein solches Prädikat definiert. Nun, auch hier bietet es sich an, die Sache zunächst umgangsprachlich anzugehen:

Ein beliebiger Term X ist der Kopf einer Liste L, falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird und – X mit K unifiziert wird.[23]

Die Zerlegung der Liste nun ist eine einfache Operation – einfach deshalb, weil dafür der vordefinierte Listenoperator '|' verwendet werden kann. Der umgangssprachliche Ausdruck wird also wie folgt in Prolog übersetzt, wobei statt der Variablen R die anonyme Variable '_' für den Rest benutzt wird (sonst kommt wieder die Meldung 'Singleton Variable'):

Prolog–Regel:	Zeilenweise Erläuterung:
kopf(X , L):- L = [K | _], X = K.	X ist der Kopf einer Liste L falls L in einen Kopf K und einen Rest (anonyme Variable) zerlegt wird[24] und X mit K unifiziert wird.

Diese Regel liefert das gewünschte Ergebnis. Entsprechend lautet auch die Regel für rest/2:

Prolog–Regel:	Zeilenweise Erläuterung:
rest(X , L):- L = [_| R], X = R.	X ist der Rest einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und X mit R unifiziert wird.

Es gibt allerdings die Möglichkeit, das Ganze ein wenig einfacher und eleganter zu formulieren. Bei diesem Verfahren geht es darum,

- die Listenzerlegung nicht explizit auszudrücken (also mit L = [K | R], Teilziel (1)); sondern diese bereits im Regelkopf zu erfassen, indem die Liste schon hier in Kopf und Rest zerlegt wird.
- auch die Unifikation von X mit K nicht explizit auszudrücken (also mit X = K, Teilziel (2)), sondern diese ebenfalls im Regelkopf zu erfassen – einfach dadurch, daß für beide dieselbe Variable verwendet wird.

Das bedeutet, daß wir statt der oben angegebenen Regel für kopf/2 stattdessen ein Fakt formulieren, welches die folgende Form hat: kopf(X, [X | _]). Dieses Fakt konstatiert die Tatsache, daß ein Term X Kopf einer Liste [X|_] ist. Dieses Verfahren, also diese vorweggenommene Auswertung nennt man PARTIELLE EVALUATION. Dazu gleich noch mehr.

Als nächstes geht es darum, die Prädikate erstes/2, zweites/2, drittes/2, die jeweils das erste, zweite und dritte Element einer Liste identifizieren. Beispielsweise soll gelten:

(A) erstes/2

Die Formulierung des ersten Prädikates, erstes/1, ist einfach – sie ist eigentlich identisch zu der Definition von kopf/2 weiter oben:

Prolog–Regel:	Zeilenweise Erläuterung:
erstes(X , L):- L = [K | _], X = K.	X ist das erste Element einer Liste L falls L in einen Kopf K und einen Rest (anonyme Variable) zerlegt wird und X mit K unifiziert wird.

Es gibt aber auch die Möglichkeit, das bereits definierte kopf/2 einzusetzen:

Prolog–Regel:	Zeilenweise Erläuterung:
erstes(X , L):- kopf(X,L).	X ist das erste Element einer Liste L falls X der Kopf von L ist

Schließlich kann auch erstes/2 durch ein Fakt ausgedrückt werden: erstes(X, [X|_]).

---

[23] 'Unifikation' bedeutet entweder, daß K an X gebunden wird, oder aber, daß K und X identisch sein müssen.

[24] Genauer gesagt: wenn L mit einer Konstruktion unifiziert wird, die entsprechend in einen Kopf und Rest zerlegt wird. Die anonyme Variable wird in den folgenden Prädikaten immer dann verwendet, wenn sie einen Term bezeichnet, auf den es sozusagen nicht ankommt, da dessen Wert für das Prädikat irrelevant ist. Andernfalls erscheint die beliebte Warnung 'Singleton Variable'.

(B) zweites/2

Das nächste Prädikat, zweites/2, scheint eigentlich im Widerspruch zu der eingangs getroffenen Äußerung zu stehen, daß man direkten Zugriff nur auf den Kopf einer Liste hat: wenn dem so ist, kann man ja garnicht das zweite (oder dritte oder vierte usw.) Element isoliert identifizieren. Aber: – und das ist ist der Kerngedanke der Listenverarbeitung – **jede Liste ist zerlegbar in Kopf und Rest**, und der Rest ist selber wieder eine Liste – hat also auch einen Kopf, auf den man wiederum Zugriff hat.

Nehmen wir zur Verdeutlichung ein Beispiel. Das Element kicks steht in der Liste [the, boy, kicks, the, ball] an der dritten Position. Wird die Liste in Kopf und Rest zerteilt, erhalten wir eine Restliste wie folgt:

```
[Kopf  |      Rest]
  =            =
[the    | [boy, kicks, the, ball]]
```

In der Restliste steht kicks an der zweiten Position. Wird diese Restliste, also [boy, kicks, the, ball], selber in Kopf und Rest aufgeteilt (hier Kopf1 und Rest1 genannt), erhalten wir folgendes Ergebnis:

```
[Kopf1  |    Rest1]
  =            =
[boy     | [kicks, the, ball]]
```

Nun steht kicks an der ersten Position. Durch die Aufteilung von Rest1, also [kicks, the, ball], in Kopf2 und Rest2:

```
[Kopf2  |    Rest2]
  =            =
[kicks   |  [the, ball]]
```

haben wir kicks in einer Position, in der wir Zugriff auf dieses Element haben – als Kopf einer Liste (schlecht ausgedrückt als Kopf des Restes des Restes des Rests).

Mit diesen Erkenntnissen im Hinterkopf ist die Problemstellung von zweites/2 nun sicher einfacher nachvollziehbar. Erstmal wieder umgangssprachlich:

Ein beliebiger Term X ist das zweite Element einer Liste L, falls diese Liste in einen Kopf K und einen Rest R zerlegt wird und der Rest R selber in einen Kopf K1 und einen Rest R1 und X mit K1 unifiziert wird.

Prolog–Regel:	Zeilenweise Erläuterung:
zweites(X , L):- L = [_ | R], R = [K1 | _], X = K1.	X ist das zweite Element einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und R in einen Kopf K1 und einen Rest (anonyme Variable) zerlegt wird und X mit K1 unifiziert wird..

Mit partieller Evaluation hat das Prädikat die folgende, einfachere Form:

Prolog–Regel:	Zeilenweise Erläuterung:
zweites(X , [_|R]):- R = [X | _].	X ist das zweite Element einer Liste [_ |R] falls X der Kopf von R ist.

Natürlich aber können wir zweites/2 auch mit Bezug auf erstes/2 ausdrücken:

Ein beliebiger Term X ist das zweite Element einer Liste L, falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird und – X das erste Element des Restes R ist.

In Prolog sieht das so aus:

Prolog–Regel:	Zeilenweise Erläuterung:
zweites(X , Liste):- L = [_ | R], erstes(X,R).	X ist das zweite Element einer Liste L falls L in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und X das erste Element von R ist.

(C) drittes/2

Auch das Prädikat drittes/2 funktioniert nachdemselben Prinzip. Das Beispiel von weiter oben ([the, boy, kicks, the ball]) hat ja bereits gezeigt, wie man das dritte Element einer Liste ermittelt:

Ein beliebiger Term X ist das dritte Element einer Liste L, falls – diese Liste in einen Kopf K und einen Rest R zerlegt wird, – der Rest R in einen Kopf K1 und einen Rest R1, – der Rest R1 in einen Kopf K2 und einen Rest – und X mit K2 unifiziert wird.

Prolog–Regel:	Zeilenweise Erläuterung:
drittes(X , L):-	X ist das dritte Element einer Liste L falls
L = [_ | R],	Liste in einen Kopf (anonyme Variable) und einen Rest R zerlegt wird und
R = [_ | R1]	R in einen Kopf (anonyme Variable) und einen Rest R1 zerlegt wird und
R1 = [K2 | _],	R1 in einen Kopf K2 und einen Rest (anonyme Variable) zerlegt wird und
X = K2.	X mit K2 unifiziert wird..

Dasselbe wieder mit partieller Evaluation:

Prolog–Regel:	Zeilenweise Erläuterung:
drittes(X , [_|R]):-	X ist das dritte Element einer Liste [_ | R] falls
R = [_ | R1],	R in einen Kopf (anonyme Variable) und einen Rest R1 zerlegt wird und
R1 = [X, _].	X der Kopf von R1 ist.

Unter Bezug auf zweites/2 schließlich könnte das Prädikat auch wie folgt formuliert werden:

Ein beliebiger Term X ist das dritte Element einer Liste [_ \|R], falls X das zweiteElement des Restes R ist.

Prolog–Regel:	Zeilenweise Erläuterung:
drittes(X , [_|R]):-	X ist das dritte Element einer Liste [_ |R] falls
zweites(X,R).	X das zweite Element von R ist.

Somit hätten wir die drei Prädikate formuliert. Um nun auch Prädikate wie viertes/2, fuenftes/2 usw. zu ermitteln, könnten wir theoretisch so weitermachen wie bisher. Was wir aber beispielsweise nicht können, ist ein Prädikat wie letzes/2, welches das letzte Element einer Liste ermittelt. Der Grund: Listen können beliebig viele Elemente enthalten – auf die oa. Weise wird aber immer nur eine endliche Zahl von Positionen 'abgearbeitet'. Es ist in der Tat so, das die o.a. Prädkate die eigentlich wirklich elegante Art der Listenverarbeitung verfehlen, nämlich die rekursive Listenverarbeitung. Diese ist der Gegenstand der nächsten Abschnitte.

5.2. Prinzipien der rekursiven Listenverarbeitung

Das Prinzip der Rekursion ist bereits in Kapitel 3 in Zusammenhang mit unterbegriff/2 eingeführt worden. Dabei ist die folgende Aussage getroffen worden:

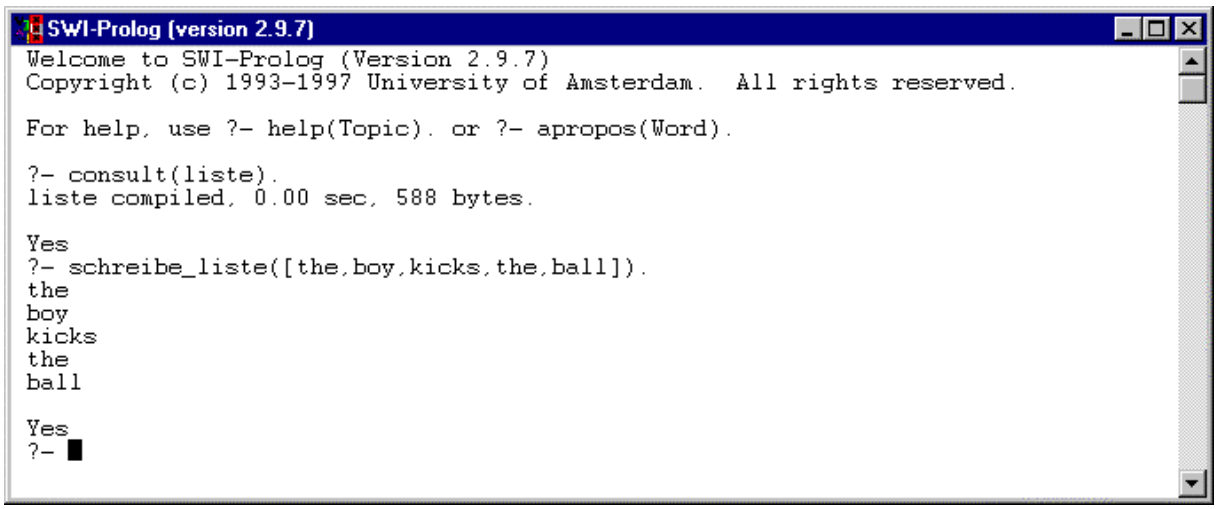
> Das Prinzip bei der Rekursion ist daher, eine einfache 'Erfolgsbedingung' zu formulieren […] und die rekursive Bedingung auf eine Art auszudrücken, daß sie quasi solange wiederholt wird, bis diese einfache Erfolgsbedingung erreicht wird.

Für die Verarbeitung von rekursiven Strukturen, also Strukturen wie Listen, bieten sich rekursive Prozeduren (Prädikate) an, die genau dieses Prinzip erfüllen. Die Prozedur wird solange durchgeführt, bis eine Erfolgsbedingung eintritt. Darunter muß hier verstanden werden, daß der Fall angegeben wird, an dem die Rekursion abbricht: bei Listen ist dieser Fall i.d.R. dann erreicht, wenn diese entweder leer sind oder nur ein Element enthalten.

Daraus ergibt sich folgendes allgemeines Verfahren zur Listenverarbeitung:

a) Wende die Prozedur auf die leere Liste (bzw. die Einerliste) an (Erfolgsbedingung
b) Zerlege die Liste in Kopf und Rest
   – Behandle den Kopf
   – Wende die Prozedur **rekursiv** auf den Rest an

Dieses Verfahren ist am besten anhand eines konkreten Beispieles zu erklären. Dazu geht es um die Formulierung eines Prädikates schreibe_liste/1, welches die Elemente einer beliebigen Liste – egal, wie lang – zeilenweise auf dem Bildschirm ausgibt. Eine Anfrage wie
schreibe_liste([the,boy,kicks,the,ball]) soll also folgendes Ergebnis bringen:



Wiederum wollen wir die Problemstellung wie folgt umgangssprachlich formulieren:

Wenn die Liste leer ist, passiert gar nichts, die Prozedur wird abgebrochen.
Wenn die Liste nicht leer, so zerlege sie in einen Kopf und einen Rest, gebe den Kopf mit write/1 auf dem Bildschirm aus, gehe mit nl in eine neue Zeile ***und wende die Prozedur auf den Rest an****.*

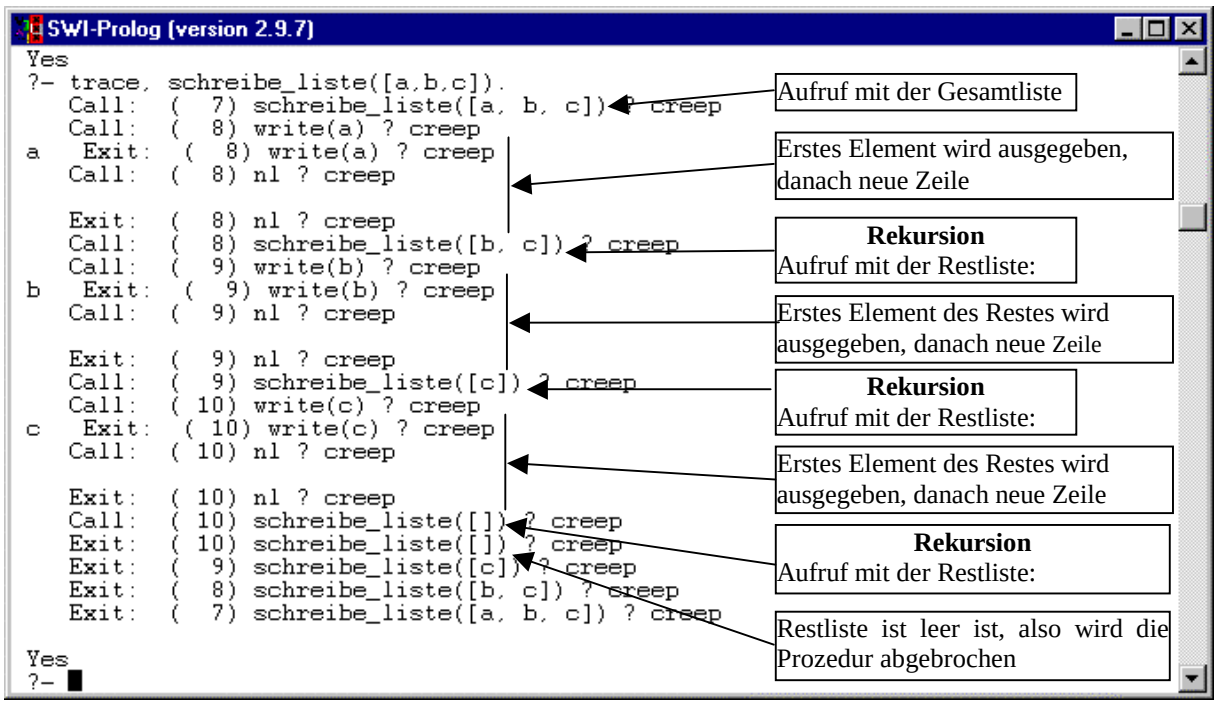
Der zentrale Punkt ist natürlich der erneute Aufruf des Prädikates mit dem Rest – der Teil, der in der Formulierung fettgedruckt ist. In Prolog hat das die folgende Form:

Prolog–Regel:	Zeilenweise Erläuterung:
schreibe_liste([ ]).	Wenn die Liste leer ist, passiert garnichts! (Abbruchbedingung)
schreibe_liste(Liste):- Liste = [K|R], write(K), nl, schreibe_liste(R).	Wenn die Liste nicht leer ist, zerlege die Liste in einen Kopf K und einen Rest R, gebe K am Bildschirm aus, gehe in eine neue Zeile rufe schreibe_liste/1 mit R auf (Rekursionsschritt!!)

Etwas einfacher, mit partieller Evaluation, sieht das Prädikat so aus:

```
schreibe_liste([ ]).
schreibe_liste([K|R]):-
   write(K),
   nl,
   schreibe_liste(R).
```

Anhand des folgenden kommentierten *Screen-Shots* wird gut deutlich, wie Prolog die gesamte Liste abarbeitet, bis schließlich die leere Liste erreicht ist und die Prozedur abbricht:



Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 2 und 3** am Kapitelende angehen. Aufgabe 3 ist detailliert erläutert.

5.2.1. Abbildung von Listen auf Listen

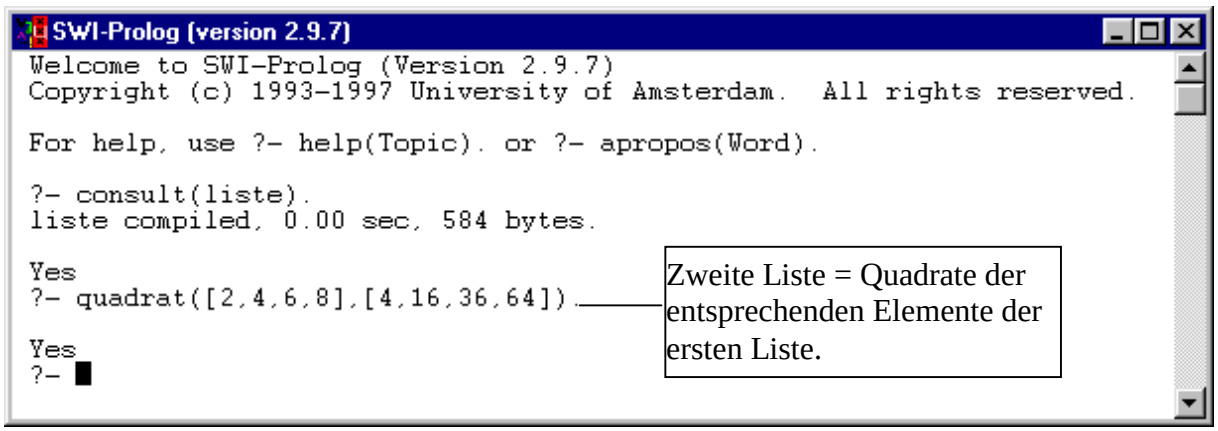
Bei den Prädikaten erstes/2, element/2, laenge/2 usw. ging es immer um eine einzelne Liste. Ein sehr häufiges Prinzip der Listenverarbeitung ist aber die Abbildung einer Liste auf eine zweite Liste (bzw. von mehreren Listen auf eine Liste). *Abbildung* bedeutet, daß ein jedes Element einer Liste L1 nach einem bestimmten Prinzip (nämlich durch eine Abbildungsfunktion) dem entsprechenden Element einer Liste L2 zugeordnet wird.

L1: [X1, X2, X3, ..., X_n]:
↓ ↓ ↓ ↓ ↓
L2: [Y1 Y2 Y3, ..., Y_n]

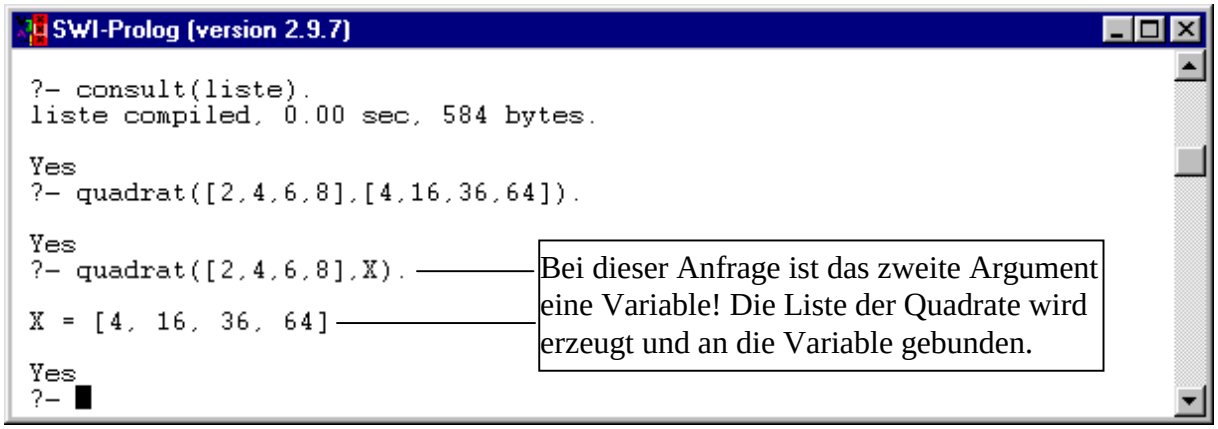
Im folgenden Beispiel bilden die Elemente der zweiten Liste jeweils das Quadrat der entsprechenden Elemente der ersten Liste. Sie sind also durch die Abbildungsfunktion Y = X*X definiert.

L1: [2, 3, 4, ..., X_n]:
↓ ↓ ↓ ↓ ↓
L2: [4 9 16, ..., X_n*X_n]

Betrachten wir dazu ein konkretes Prädikat, nämlich quadrat/2, welches genau diese Aufgabe erfüllt: es hat als Argumente zwei Listen, die Elemente der zweiten Liste bilden jeweils das Quadrat der entsprechenden Elemente der ersten Liste:



Die Bezeichnung 'Abbildung' ist nun vielleicht insofern mißverständlich, als sie möglicherweise suggeriert, beide Listen würden bereits vorgegeben sein – so wie gerade in diesem Beispiel, bei welchem quadrat/2 mit zwei instantiierten Argumenten aufgerufen wurde (d.h. es wurden keine Variablen verwendet). Interessant wird die Sache aber insbesondere dadurch, daß Prädikate, die Listen aufeinander abbilden, auch verwendet werden, um Listen sozusagen neu aufzubauen – wie im nachfolgenden Screen-Shot, bei welchem die zweite Liste (die sogenannte ZIELLISTE) erst ermittelt werden soll:



Das Verfahren, das zwei Listen aufeinander abbildet, arbeitet völlig analog zu dem der allgemeinen rekursiven Listenverarbeitung:

1. Formuliere eine Endbedingung (z.B.∴ beide Listen sind leer)
2. Zerlege beide Listen in Kopf und Rest
   a. Wende die Abbildungsfunktion auf die Listenköpfe an;
   b. Wende die Prozedur rekursiv auf die Listenreste an.

Zur Illustration dient ein Beispiel, nämlich genau das Prädikat quadrat/2:

> Definiert ein Prädikat quadrat/2, das dann erfolgreich ist, wenn jedes Element y_i von Liste2 das Quadrat des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot x_i$.

Problembeschreibung:

> *Wenn die Ausgangsliste leer ist, ist auch die Zielliste leer.*
>
> *Zerlege die Ausgangs- und Zielliste jeweils in Kopf und Rest.*
>
> > *Der Kopf der Zielliste ist das Quadrat des Kopfes der Ausgangsliste.*
> >
> > *Der Rest der Zielliste ist die 'Quadrat'-liste des Restes der Ausgangsliste.*

Diese Problembeschreibung wird direkt in Prolog übersetzt:

```
quadrat([ ], [ ]).
quadrat ([K1 | R1], [K2 | R2]):-
        K2 is K1 * K1,
        quadrat(R1, R2).
```

Die erste Regel (das erste Fakt) ist die Endbedingung – hier also die leeren Listen. In der rekursiven Regel wird zunächst der Kopf 'bearbeitet', und zwar nach der Abbildungsfunktion $Y = X \cdot X$, dann wird das Prädikat erneut mit den Restlisten aufgerufen.

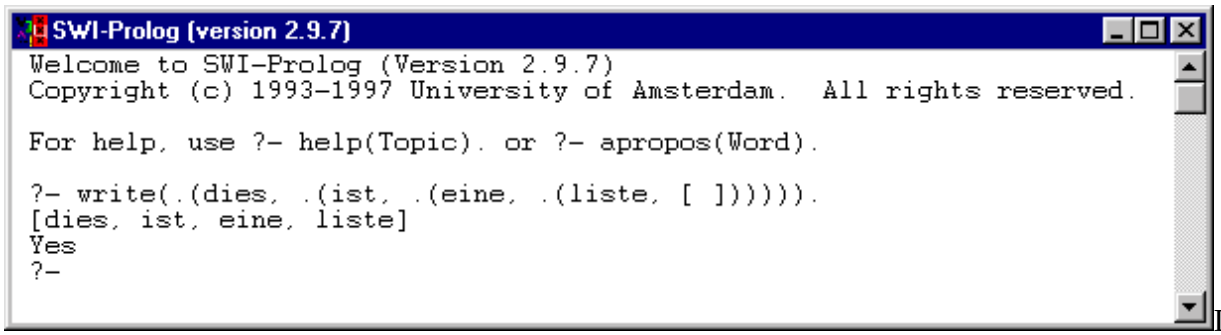
Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt die **Aufgaben 5 – 7** am Kapitelende angehen.

Übungen zu Kapitel 5

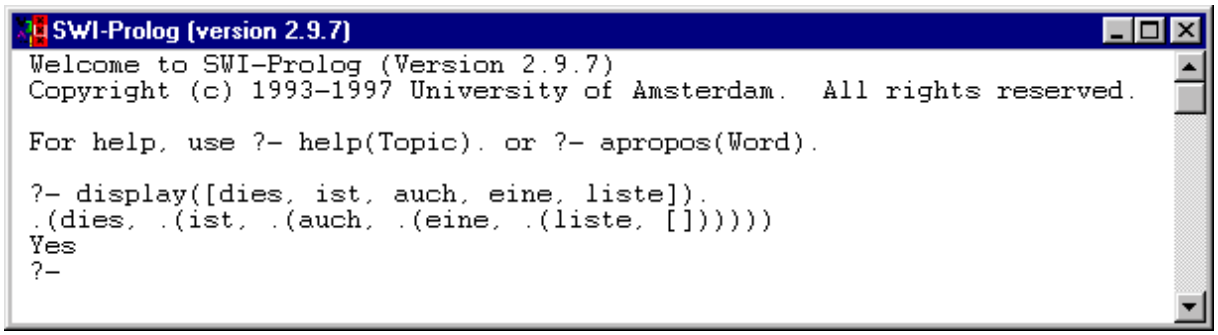
Aufgabe.1.

Übersetzt (auf Papier) die folgenden Listen (einfache Schreibweise) in 'explizite' Listen (mit Listenoperator und leeren Listen) einerseits und in Funktor-Argument-Strukturen andererseits:
[the, boy, slept]
[a, b, [1,2], c] (Liste mit eingebetteter Liste).
Übersetzt (auf Papier) die folgende Funktor-Argument-Struktur in eine Liste mit einfacher Schreibweise:
.(der, .(junge, .(lacht, [ ])))

Ein kleiner Tip: man kann diese Übung mit Prolog überprüfen. Das Prädikat write/1 nämlich kann dazu benutzt werden, Funktor-Argument-Strukturen in Listen zu 'übersetzen':



Das Prädikat display/1 dagegen kann eingesetzt werden, um für einen Term (hier also eine Liste) anzuzeigen, wie dieser intern in Prolog repräsentiert wird:



Die nächsten beiden Übungen sind rekursive Prädikate zur Listenverarbeitung. Auch hier ist es so, daß der Hauptnutzen der Übungen darin liegt, ein Gefühl für den Umgang mit Prolog–Listen zu entwickeln – der tatsächliche Nutzen der zu erstellenden Prädikate für spätere Programme ist also eher mittelbar.

Aufgabe 2

Definiert ein Prädikat element/2. Das erste Argument ist ein Term, das zweite Argument eine Liste. Das Prädikat element/2 soll für den Term ermitteln, ob er Element in der Liste ist oder nicht. Beispielsweise soll gelten:
element(c, [a,b,c,d]).

Lösungstip: der 'einfache' Fall, die Erfolgsbedingung in diesem Beispiel ist nicht mit Bezug auf die leere Liste formuliert, sondern die Tatsache, daß der Kopf einer Liste natürlich Element dieser Liste ist. Man könnte es so ausdrücken:

Ein Term X ist Element einer Liste L, wenn diese in Kopf und Rest zerlegt wird und X der Kopf von L ist.

Ein Term X ist Element der Liste L, wenn es Element des Restes ist.

Durch eine entsprechende Prolog-Formulierung wird die Liste solange abgearbeitet, bis irgendwann der Fall eingetreten ist, bei welchem diejenige Restliste erreicht wird, deren Kopf das gesuchte Element X ist (so es denn tatsächlich in der Liste vorkommt). Übrigens hat dieses Prädikat das Fließmuster element(?Term,+Liste) (zu Fließmuster siehe Kapitel 4, *Notationskonventionen für Prädikate*). Das heißt, daß bei einer Anfrage das erste Argument von element/2 auch eine Variable sein kann. Dann gibt Prolog alle Werte aus, an die diese Variable gebunden werden kann – mithin alle Elemente der Liste:

Aufgabe 3

> Definiert ein Prädikat letztes/2, welches das letzte Element einer Liste ermittelt. Das erste Argument ist ein Term, das zweite Argument eine Liste. letztes/2 soll für den Term ermitteln, ob er das letzte Element in der Liste ist oder nicht. Beispielsweise soll gelten:
> letztes(d, [a,b,c,d]).

Auch diese Aufgabenstellung ist nur durch die rekursive Listenverarbeitung zu bewältigen – einfach deshalb, weil man nie genau weiß, wieviele Elemente die Liste aufweist, und also keine Prädikate formulieren kann, die das Listenende in einer vorher festgelegten Zahl von Schritten ermitteln.

Der einfache Fall, die Erfolgs- oder Abbruchbedingung ist in diesem Prädikat die EINERLISTE. Diese wird wie folgt notiert: [X] (siehe dazu auch die Definition von Liste im letzten Kapitel). Umgangsprachlich wird die Problemstellung wie folgt umschrieben:

Wenn eine Liste nur ein Element aufweist, so ist dieses Element das letzte Element der Liste.

Wenn eine Liste mehr als ein Element aufweist, so zerlege sie in Kopf und Rest. Das gesuchte Element ist das letzte Element des Restes.

Komplexere Fälle der Listenverarbeitung entstehen, wenn zusätzliche Argumente mitgeführt werden. Ein solcher Fall kann beispielsweise dann eintreten, wenn die Länge einer Liste ermittelt werden soll – die Frage ist, wie das geht, denn dafür muß Prolog die Elemente der Liste ja irgendwie 'zählen':

Aufgabe 4

> Definiert ein Prädikat laenge/2, welches die Länge einer Liste ermittelt. Das erste Argument ist eine Liste, das zweite Argument ist eine Zahl, die die Länge der Liste angibt. Beispielsweise soll gelten:
> laenge([a,b,c,d,e],5).

Für dieses Prädikat benötigen wir entsprechende Operatoren, in diesem Fall den Additionsoperator '+' und den Auswertungsoperator 'is'. Dazu erstmal ein paar Anmerkungen.

Exkurs: Auswertung arithmetischer Ausdrücke

In Standard-Prolog sind eine Reihe von arithmetischen Operatoren definiert: +, –, *, / etc. die in entsprechenden Operator-Operand-Strukturen verwendet werden können, z.B. 2 + 3, 5 – 2, A * B, N / 3, etc. Dabei muß man jedoch berücksichtigen, daß es sich hier um bloße syntaktische Konstruktionen handelt, die nicht automatisch ausgewertet werden. Durch den Ausdruck N = 2 + 3 wird N nicht etwa mit

der Zahl 5 unifiziert, sondern mit der Struktur 2 + 3 (≡ '+'(2, 3)). Wenn arithmetische Ausdrücke ausgewertet werden sollen, muß ein eigener Auswertungsoperator is/2 verwendet werden: Ausdruck1 is Ausdruck2. Dieser Operator ist so definiert, daß Ausdruck2 ausgewertet wird und das Ergebnis mit Ausdruck1 unifiziert wird. Ausdruck1 ist entweder eine Variable oder eine Ganzzahl, Ausdruck2 muß ein syntaktisch korrekter arithmetischer Ausdruck sein.

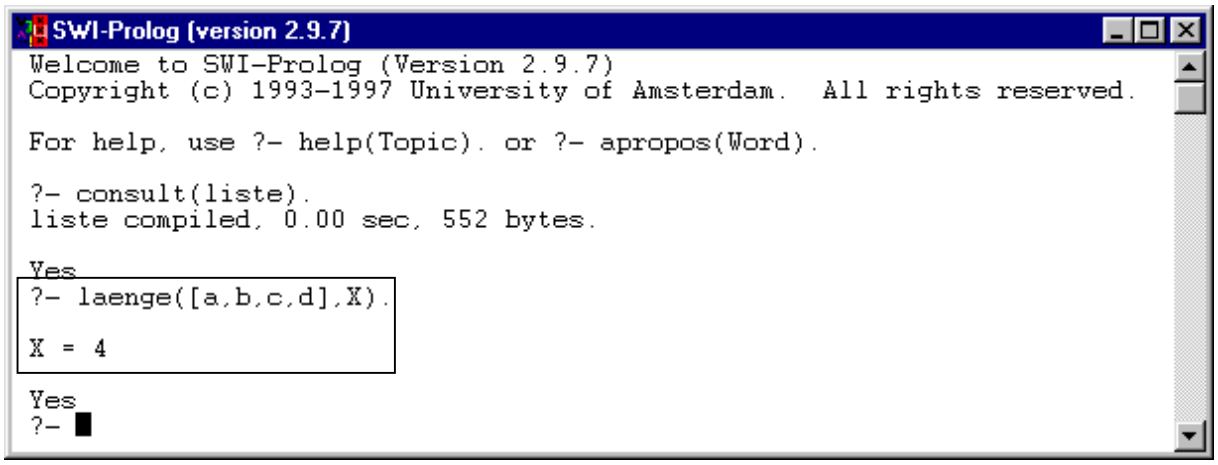
?- N is 2+3
Auswertung

N = 5

?- 5 is 2 + 3
Yes

Wichtig: Ein arithmetischer Ausdruck kann nur dann ausgewertet werden, wenn er keine ungebundenen Variablen enthält.

Damit können wir zu unserer Aufgabenstellung zurückkehren, die Länge einer Liste zu ermitteln.Das Prädikat laenge/2 soll also folgendes leisten: eine Anfrage wie laenge([a,b,c,d],X) führt dazu, daß Prolog die Länge der Liste ermittelt und X an den entsprechenden Wert bindet:



Um uns dem Verfahren, welches dafür angewendet wird, zu nähern, betrachten wir zunächst die folgenden Listen. Die erste Liste ist leer – ihre Länge ist entsprechend null. Die zweite Liste enthält zwei Elemente, die dritte Liste drei und die vierte Liste fünf – wobei das vierte Element selber eine Liste ist:

[ ]	Länge = 0.
[a, b]	Länge = 2.
[a, b, c]	Länge = 3.
[a, b, [1, 2, 3], c]	Länge = 4.

Der entscheidende Fingerzeig für die Beschreibung des Problemes kann nun wie folgt durch die Aufteilung der Listen in Kopf und Rest dargestellt werden:

[ ]	Länge = 0.
[a | [b]]	Länge = 2 – bzw. 1 + 1
[a | [b, c]]	Länge = 3 – bzw. 1 + 2
[a | [b, [1, 2, 3], c]]	Länge = 4 – bzw. 1 + 3

Wir sehen: die Länge einer Liste ist immer um eins größer als die Länge der Restliste. Die Länge der leeren Liste ist immer null. Genau diesen Sachverhalt gilt es, in Prolog umzusetzen:

Die Länge der leeren Liste ist null.

Die Gesamtlänge der Elemente einer Liste [K | R] ist um eins größer als die Gesamtlänge der Restliste R., d.h. ***Gesamtlänge = Restlänge + 1****.*

Salopp könnte man es so sagen: *Wenn ich weiß, wie lang die Restliste ist, weiß ich auch, wie lang die ganze Liste ist*. Diese Problembeschreibung läßt sich direkt in Prolog übersetzen. Für die Formulierung benötigen wir die beiden arithmetischen Operatoren is/2 und +/2.

```
laenge([ ],0).
laenge([K | R],X):-
        laenge(R,Y),
        X is Y + 1.
```

Das klingt ja zu schön, um wahr zu sein – auch hier mag man sich, ähnlich wie bereits bei der Formulierung der Prädikate unterbegriff/2, vorfahr/2 usw., fragen, wie das den funktionieren soll, woher also Prolog wissen kann, wie lang die Restliste ist? Schließlich wollen wir das Prädikat laenge/2 ja erst definieren, verwenden es aber hier bereits im Regelrumpf der rekursiven Regel, um die Länge des Restes zu ermitteln.

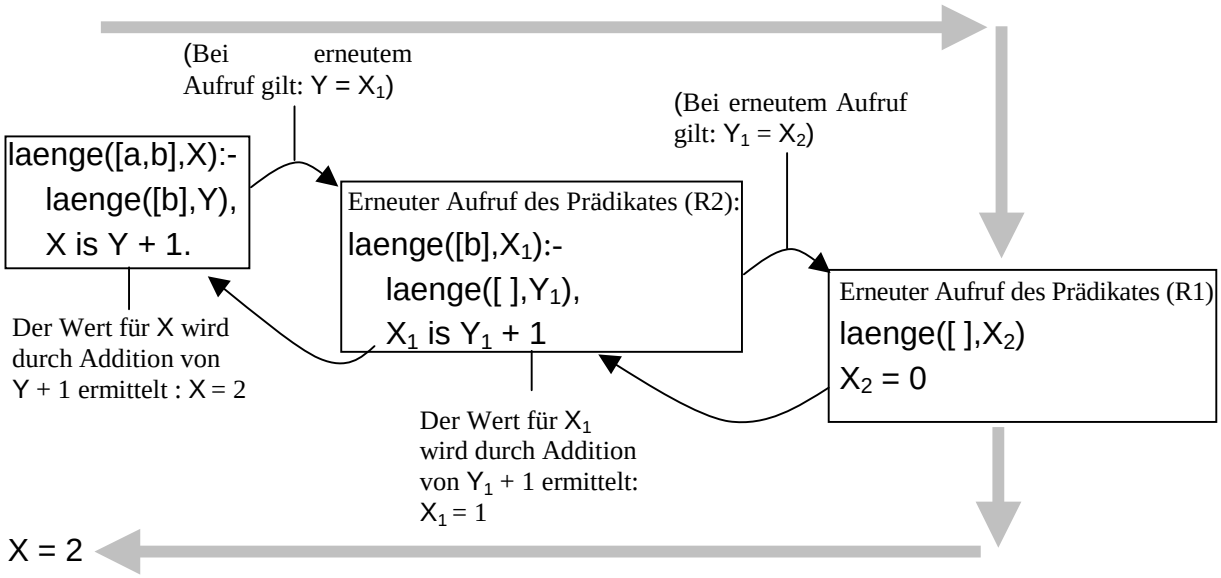
Was also passiert, wenn dieses Prädikat mit einem entsprechenden Argument aufgerufen wird? Nun, ist dieses Argument eine leere Liste, so ist die Antwort schnell gefunden:

?- laenge([ ],X). Da diese Liste leer ist, kommt Regel 1 zur Anwendung:

X = 0

Ist die Liste aber nicht leer, wird das Prädikat erneut aufgerufen – diesmal mit dem Rest der Liste. Schematisch könnte das wie folgt dargestellt werden, die Variablen sind mit Indizes versehen, da auch Prolog die Variablen bei erneutem Aufruf des Prädikates umbenennt.

?- laenge([a,b],X). Da diese Liste nicht leer ist, Aufruf von R2:

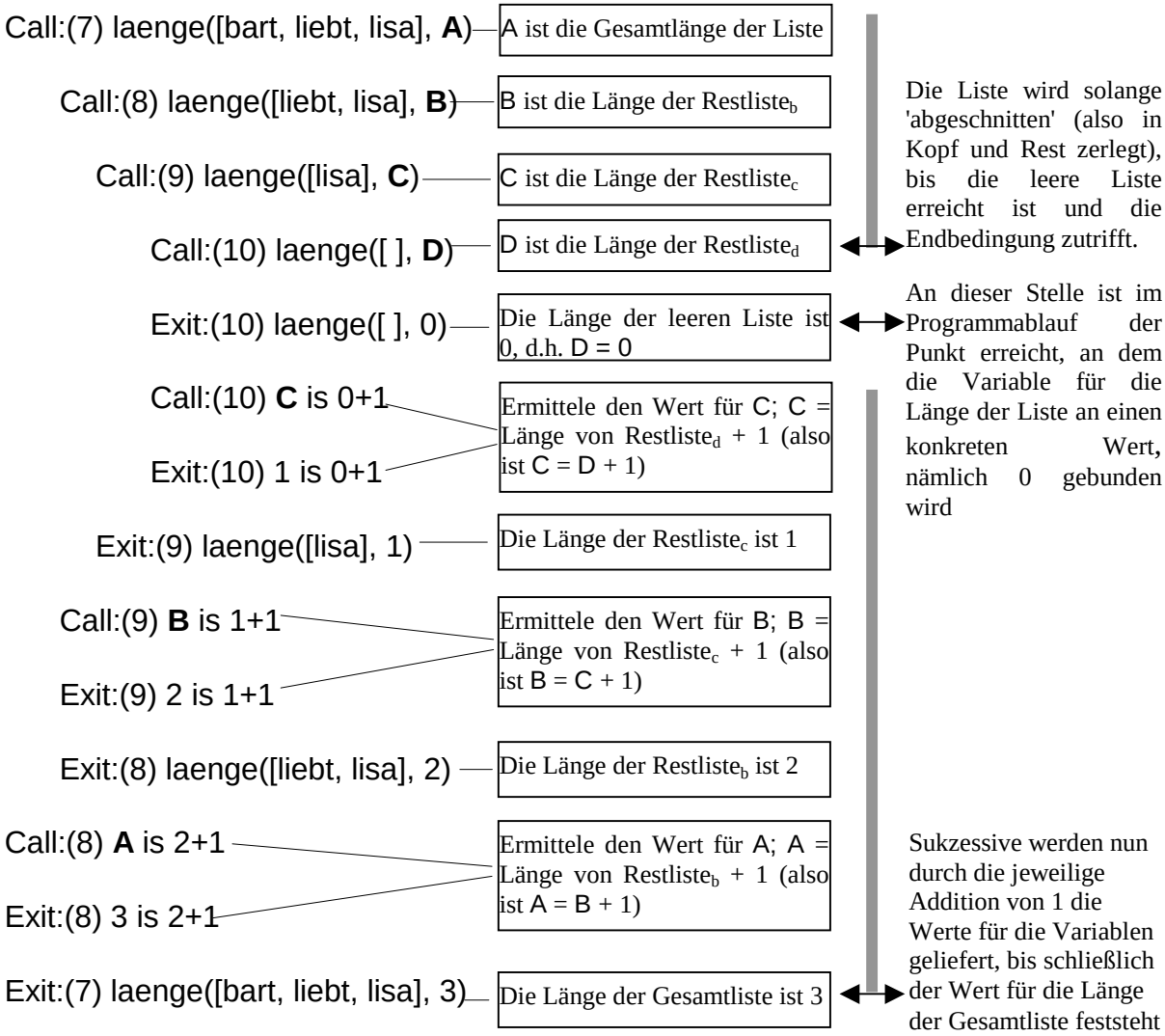


(Kleine Anmerkung: falls diese Graphik den Geist mehr ver- als entwirrt – akzeptiert)

Prolog ruft das Prädikat solange auf, bis der einfache Fall eingetreten ist – bis also die leere Liste erreicht wird. Danach geht es quasi wieder zurück – angefangen bei Null werden die Werte für die Längen der jeweiligen Restlisten durch die sukzessive Addition von Eins zu den Restlistenlängen (auahauaha!) ermittelt.

Ein Blick auf die nachfolgende Darstellung soll ebenfalls Erleuchtung bringen, indem gezeigt wird, wie Prolog eine entsprechende Anfrage abarbeitet. Es handelt sich um eine leicht modifizierte Auflistung der Ausgabe, die Prolog liefert, wenn die Anfrage laenge([bart, liebt, lisa],A). mit der *Trace*-Funktion aufgerufen würde. Sie ist insofern modifiziert, als die jeweiligen Variablen, deren Wert es zu ermitteln gilt, nicht als _L112 oder _G4301 dargestellt sind (wie bei der Original-Trace-Funktion), sondern von A–D gegliedert sind. Es lohnt sich, in den sauren Apfel zu beißen und diese zunächst etwas unübersichtlich scheinende Darstellung genau durchzuarbeiten – ihr ist nämlich genau zu entnehmen, wie die Verarbeitung dieses rekursiven Listenprädikates funktioniert.

*?- trace, laenge([bart,liebt,lisa], **A**).*



X = 3

Yes

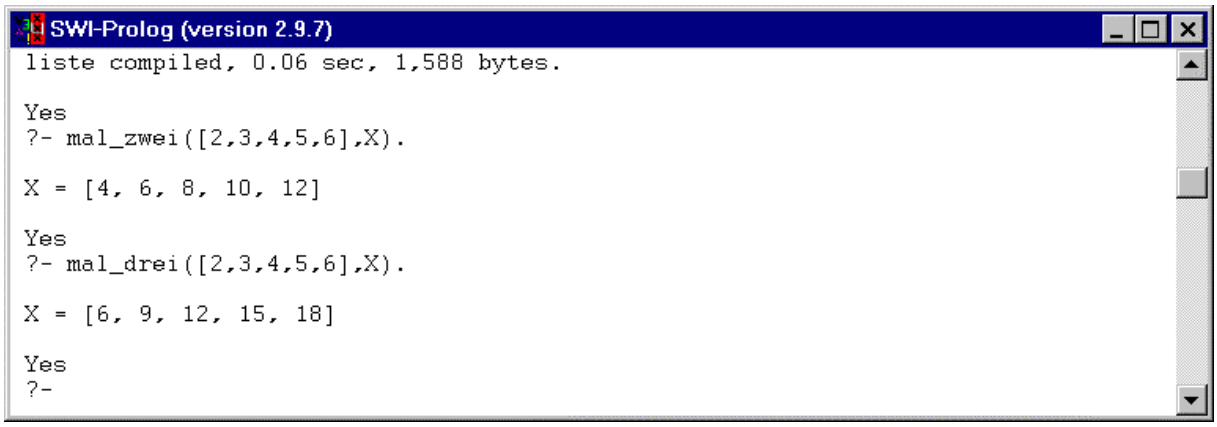
Man sieht, was passiert: Prolog arbeitet die Liste sukzessive ab (d.h. zerlegt sie in Kopf und Rest), und – das ist ein ganz wichtiger Punkt – legt die nicht-beantwortbaren Ziele, hier also die Werte der Variable A – C, solange auf Eis, bis schließlich die Endbedingung erreicht ist, und ein konkreter Zahlenwert für die Variable D ermittelt ist. Danach werden die zuvor zurückgestellten Ziele mit dem Additionsverfahren nacheinander abgearbeitet, bis endlich die Länge der Gesamtliste ermittelt ist.

Soviel zur internen Verabeitung rekursiver Listenprädikate. In vielen gängigen Prolog-Einführungen wird dieser Punkt nicht explizit aufgegriffen – denn dem eigentlichen Ziel aller Einführungen, der Vermitlung der deklarativen Programmierung, bei welcher ein Programm aus einer reinen Problembeschreibung besteht (wie in diesem Skript immer die kursiv gedruckten Textstellen), dessen Lösung dann Prolog obliegt, trägt die explizite Angabe der dabei durchgeführten Verarbeitungsschritte nicht unbedingt bei. Im vorliegenden Skript wird aber dennoch versucht, zu zeigen, was intern passiert – einfach deshalb, weil erfahrungsgemäß im Zusammenhang mit rekursiven Prädikaten immer wieder die Frage 'wie kann das denn funktionieren' aufkommt. Aber auch hier, bei der rekursiven Listenverarbeitung, verhält es sich wie im Abschnitt über rekursive Prädikate im vierten Kapitel: das eigentliche Verarbeitungsverfahren ist doch irgendwie einfacher, als es die komplizierten Ausführungen vermuten lassen. Entsprechend wird auf die animierte Darstellung in der Veranstaltung verwiesen, die dann hoffentlich zum weiteren Verständnis dieser eleganten Programmiertechnik beiträgt.

Aufgabe 5

Definiert die Prädikate mal_zwei/2, mal_drei/2 und mal_vier/2, die dann erfolgreich sind, wenn jedes Element y_i von Liste2 entweder das Doppelte, das Dreifache oder das Vierfache des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot 2$, $y_i = x_i \cdot 3$ und $y_i = x_i \cdot 4$.

Diese Prädikate sollen Anfragen wie die folgenden ermöglichen:



Problembeschreibung für mal_zwei/2:

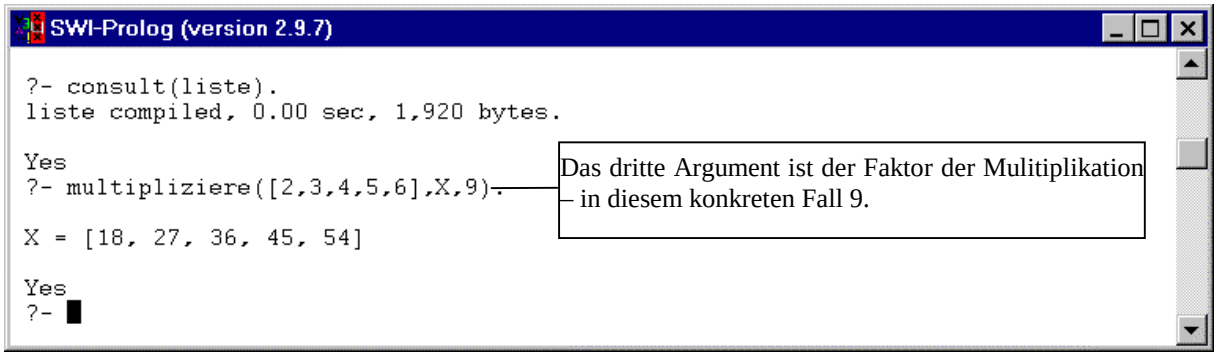
Wenn die Ausgangsliste leer ist, ist auch die Zielliste leer.

Zerlege die Ausgangs- und Zielliste jeweils in Kopf und Rest.

Der Kopf der Zielliste ist das Doppelte des Kopfes der Ausgangsliste.

Der Rest der Zielliste ist die Liste der doppelten Elemente des Restes der Ausgangsliste.

Diese drei Prädikate sind im Grunde ja identisch – bis auf den Faktor der Mulitplikation, dieser ist entweder zwei, drei oder vier. Schöner wird das Ganze, wenn der Benutzer diesen Faktor bei der Anfrage selber bestimmen kann – z.B. so:



Aufgabe 6

Definiert ein Prädikat multipliziere/3, also multipliziere(Liste1, Liste2, N), welches dann erfolgreich sind, wenn jedes Element y_i von der Zielliste das N-fache des entsprechenden Elementes x_i von Liste1 ist, d.h. $y_i = x_i \cdot N$.

Aufgabe 7

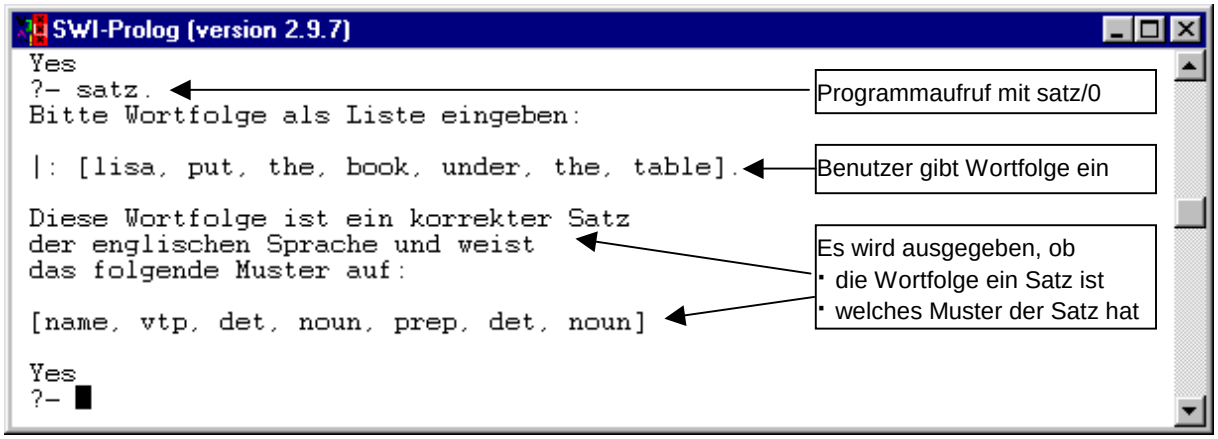
Legt in einer Datei lexikon.pl eine Reihe von Fakten lex/2 über die Zugehörigkeit von Wörtern zu lexikalischen Kategorien an (z.B. lex(kicks,verb)). Schreibt ein Prädikat lexkat/2, das jedem Wort einer Liste von Wörtern seine Wortklasse zuordnet. Beispielsweise soll gelten:
lexkat([the, boy], [det, nomen])

Kapitel 6.
Satzanalyse mit Prolog

In diesem Kapitel geht es um die Verarbeitung von natürlichsprachigen Sätzen mit Prolog. Wir wollen ein Programm entwickeln, welches in der Lage ist, für eine beliebige englische Wortfolge anzugeben, ob diese ein Satz der englischen Sprache ist oder nicht, und wenn ja, diesem Satz eine oder (bei struktureller Ambiguität) mehrere Strukturbeschreibungen zuordnet. Dieses Programm wird in kleinen Schritten aufgebaut, so daß die endgültige Form das Resultat von jeweils nachvollziehbaren Modifikationen ist. Auf diese Weise soll erreicht werden, daß am Ende solch einer sukzessiven Programmentwicklung nicht nur ein lauffähiges Programm steht, sondern auch ein Grundverständnis für die spezifischen Probleme und Methoden geschaffen wird, die für diesen Bereich der Computerlinguistik relevant sind.

6.1. Teil 1: Akzeptor I: Satzanalyse mit Mustervergleich (*Pattern–Matching*)

In diesem Abschnitt geht es darum, eine Reihe von Satz-Strukturmustern in die Prolog-Datenbank zu schreiben, mit denen eine beliebige Wortkette 'abgeglichen' wird. Die erste Version ist ein Programm, welches folgendes zu leisten vermag:



Das Programm wird mit satz/0 aufgerufen. Die Benutzer können daraufhin eine Wortkette englischer Wörter in Form einer Liste eingeben. Wenn diese Liste mit den Datenbanksatzstrukturmusterlisten (Hottentottenstottertrottelmutter) übereinstimmt, wird die Wortfolge als englischer Satz akzeptiert, und darüberhinaus wird das Satzstrukturmuster ausgegeben. Einen so gearteten Parser nennt man ERKENNER oder AKZEPTOR.

6.1.1. Grundlagen

Bevor wir diese Problemstellung in einem Programm umsetzen, wollen wir die Sache zunächst unabhängig von Prolog angehen. Letztendlich geht es um nichts anderes, als eine (Mini-)Grammatik zu formulieren. 'Grammatik' soll hier im engen Sinne verstanden werden als eine Theorie über eine Einzelsprache, welche festlegt, welche Ketten von Wörtern dieser Sprache grammatische Ketten (z.B Sätze) dieser Sprache sind.[25] Eine wichtige Eigenschaft von Sätzen ist natürlich die lineare Anordnung der einzelnen Wörter – *Milben leben in Kopfkissen ist* o.k., **in leben Milben Kopfkissen* dagegen nicht. Wenn man von konkreten Instanzen der lexikalischen Kategorien abstrahiert und also sich nicht auf konkrete Wörter, sondern auf Wortarten bezieht, funktioniert eine einfache Grammatik schlichtweg so, daß eine Reihe von Anordnungsmustern für lexikalische Kategorien vorgegeben wird. Eine Wortkette muß dann also in eine entsprechende Anordnung lexikalischer Kategorien übersetzt werden, und diese muß mit den Mustern verglichen werden. Nur solche Anordnungen, die in den Mustern erfaßt sind, werden als korrekte Sätze identifiziert.

[25] Ein wohlbekanntes Beispiel für eine solche Art von Grammatik ist z.B. die in der Syntaxeinführung vorgestellte Phrasenstrukturgrammatik.

Betrachten wir dazu eine konkrete kleine Grammatik, die völlig frei ist von linguistischen Finessen und also entsprechend leicht zu falsifizieren, da hier zunächst einmal das Prinzip deutlich werden soll. Es gelten die folgenden Abkürzungen (an die wir uns am besten gleich hier gewöhnen, da wir sie auch in dem Programm benutzen wollen):

Wortart	*Bezeichnung*	*Beispiel*
Determinatoren	det	*the*
Eigennamen	name	*Bart*
Nomen	noun	*dog*
intransitives Verb	vi	*slept*
transitives Verb	vt	*kissed*
bitransitives Verb mit PP–Ergänzung	vtp	*put*
Präposition	prep	*on*
Adjektiv	adj	*ugly*
Adverb	adv	*very*

'Wortart' soll hier im Sinne von Lexemklasse oder lexikalischer Kategorie verstanden werden. Angaben darüber, über welche Elemente eine bestimmte Lexemklasse realisiert werden kann, finden sich im Lexikon. Wir gehen zunächst von dem folgenden kleinen Lexikon aus:

det → {the, a, these}
name → {bart, lisa, john, mary}
noun → {girl, girls, boy, book, books, dog, table, garden}
vi → {slept, cried, died}
vt → {loved, kicked, kissed, painted}
vtp → {gave, put, sold }
prep → {to, in, on, under, into}
adj → {ugly, beautiful}
adv → {very, terribly}

Als potentielle Satzstrukturmuster für das Englische kommen z.B. die folgenden in Frage:

1) [det, noun, vi]
2) [name, vi]
3) [det, noun, vt, det, noun]
4) [det, noun, vt, name]
5) [name, vtp, det, noun, prep, det, noun]
6) [det, noun, vtp, det, noun, prep, name]
7) [det, adv, adj, noun, vt, det, noun]

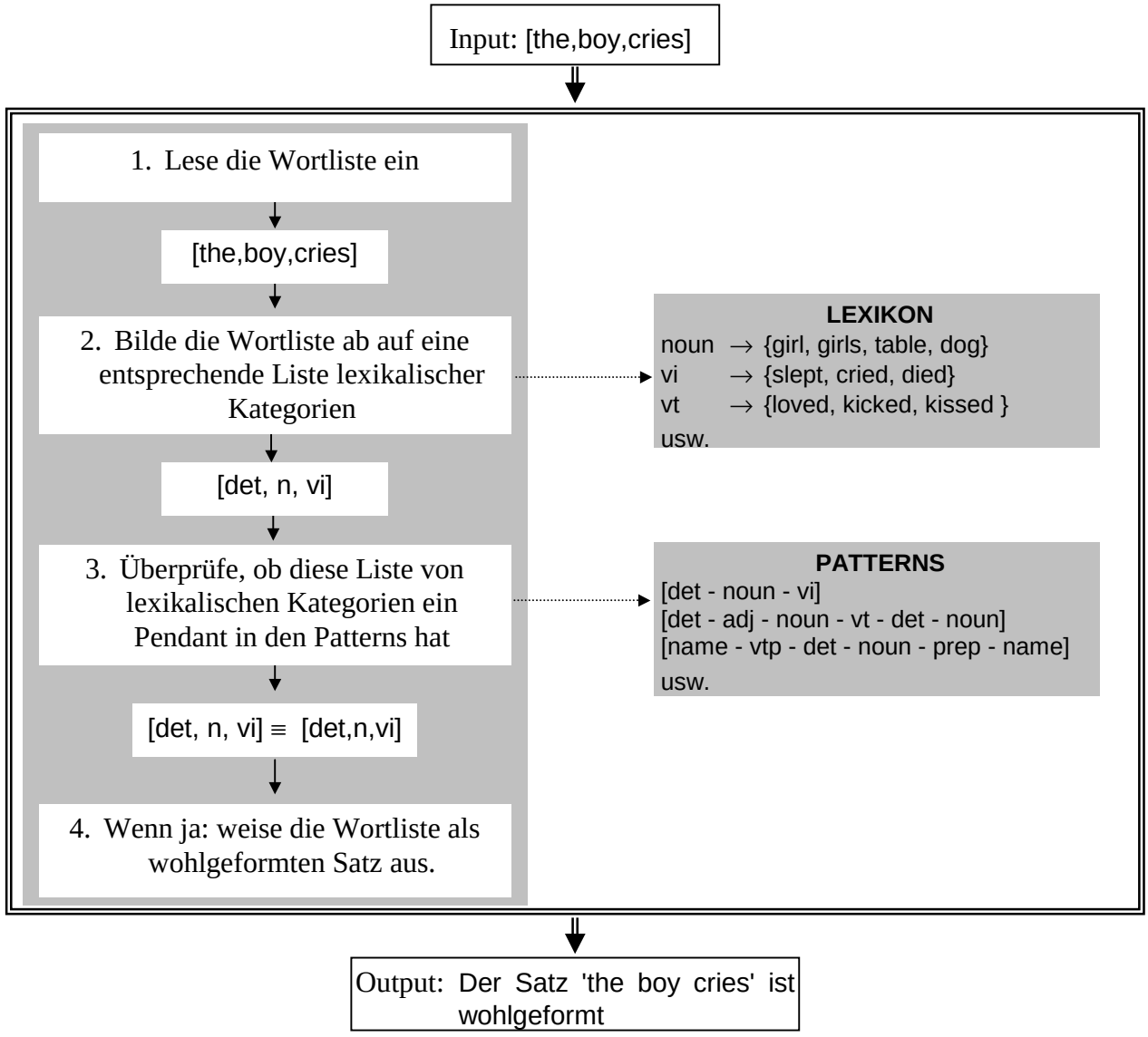
Über diese einfache Grammatik können nun unter anderem die folgenden Wortketten als grammatische Folgen, sprich Sätze des Englischen abgeleitet werden:

Lisa slept, John cried (Muster 2), *The girl loved the dog, A boy kicked the table, These girls kissed Bart* (Muster 3), *Mary gave the dog to the girls, Bart put the book under the table* (Muster 5), *The girls sold the books to John* (Muster 6) *a terribly ugly boy kicked the dog* (Muster 7) und so weiter. Natürlich würde zunächst auch die Kette **these boy died* als korrekt erkannt – dazu später noch mehr.

Die Umsetzung dieser Grammatik in Prolog ist einfach. Der Mini-Pattern-Matcher soll zunächst die folgenden Komponenten umfassen:

- Angaben darüber, welcher Wortart ein Wort angehört (Lexikon)
- Angaben darüber, welche Satzmuster wohlgeformt sind (Patterns)
- Eine Funktion, die eine Eingabeliste von Wörtern auf eine entsprechende Liste lexikalischer Kategorien abbildet (Bezug: das Lexikon)
- Ein Funktion, die eine Liste lexikalischer Kategorien mit den (in Listenform notierten) Patterns vergleicht. (Bezug: die Patterns)

Ausgehend von der Überlegung, dass die Benutzer die zu analysierende Wortkette in Form einer Liste eingeben, kann Programmaufbau und -ablauf wie folgt charakterisiert werden:



6.1.2. Die Realisierung des Mini-Pattern-Matchers in Prolog

Was die Umsetzung betrifft, sollen zuerst einmal die folgenden Maßgaben gelten:

- das Lexikon soll in einer eigenen Datei namens **lexikon.pl** erstellt werden.
- die Patterns sollen in einer eigenen Datei namens **patterns.pl** erstellt werden.
- der 'Steuermechanismus', sprich diejenigen Komponenten, die in der Graphik in den Punkten 1.-4. notiert sind, soll in einer eigenen Datei namens **grammatik1.pl** erstellt werden.

Bei diesem kleinen Programm mag man sich fragen, wieso die Komponenten nicht zusammen in einer einzigen Datei erfaßt werden. Die Antwort darauf lautet, dass wir beispielsweise das Lexikon auch für weitere Satzanalyseprogramme verwenden werden, und es sich also nicht nur wegen der besseren Transparenz des Programmes anbietet, die einzelnen Komponeten voneinander zu trennen, sondern dieses auch ganz praktische Gründe hat.

6.1.2.1. Das Lexikon

Die Zugehörigkeit einzelner Wörter zu bestimmten Lexemklassen wird in Form von lex/2-Fakten notiert: lex(Wort,Kat), also z.B.

```
lex(the,det).
lex(boy,noun).
```

usw.

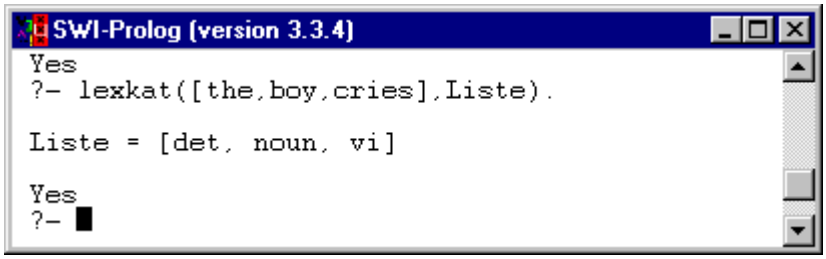
6.1.2.2. Die Patterns

Die Satzmuster werden in Form von muster/1-Fakten notiert; das Argument von muster/1 ist eine Liste: muster(Liste), also z.B.
muster([det,n,vi]).
muster([name,vt,det,noun]).
usw.

6.1.2.3. Der Pattern-Matcher

Für die Steuerdatei benötigen wir zunächst das Hilfsprädikat lexkat/2.

lexkat/2 (lexkat(Liste1,Liste2)) bildet eine Liste von Wörtern ab auf eine entsprechende Liste lexikalischer Kategorien:



Mithilfe von lexkat/2 kann der Pattern-Matcher über ein Prädikat satz/0 wie folgt realisiert werden:

1. fordere den Benutzer auf, eine Wortkette in Form einer Liste einzugeben (write...)
2. lese die Liste ein (read...)
3. bilde die Liste ab auf eine Liste lexikalischer Kategorien (lexkat...)
4. überprüfe, ob diese Liste einem Satzmuster entspricht (muster...)
5. wenn ja, weise den Satz als wohlgeformt aus und zeige das Satzmuster an. (write...)

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt **Aufgabe 1** am Kapitelende angehen.

Damit hätten wir ein kleines Programm für einen Satzakzeptor erstellt. Wie aber sehr leicht nachzuweisen ist, ist dieses Programm wirklich nur eine erste Annäherung an komplexere Programme. Die Mängel sind unterschiedlicher Natur; können wie folgt beschrieben und sollen der Reihe nach revidiert werden:

- Das Programm erkennt auch ungrammatische und anomale Formen als korrekt - beispielsweise die Ketten **a girls kissed these boy* oder ?*the book cried*.
- Das Programm rekurriert – wie alle Grammatikprogramme – stark auf das Lexikon. Falls nun in einem zu analysierenden Satz ein Wort auftritt, welches nicht im Lexikon erfaßt ist, kann die Analyse nicht vollzogen werden. Wieviel angenehmer wäre es doch, wenn in einem solchen Fall durch eine entsprechende Benutzerschnittstelle das Programm in der Lage wäre, das Lexikon um genau das fehlende Wort zu ergänzen, und die Analyse des Satzes doch durchgeführt würde.
- Die Satzstrukturmuster sind allesamt 'per Hand' eingegeben, nicht aber aus einer vernünftigeren Grammatik abgeleitet. Dieser Punkt ist ein ganz zentrales Manko des Programmes. Was in den Mustern erfaßt wird, ist die reine lineare Anordnung der lexikalischen Kategorien. Dass Sätze aber hierarchische Strukturen sind, kann in der Grammatik, die überhaupt keinen Bezug nimmt zu syntaktischen Kategorien (also NP, VP etc.) überhaupt nicht zum Ausdruck gebracht werden. Damit würde beispielsweise die Mehrdeutigkeit des Satzes *Lisa watched the boy with the telescope* über die Zuordnung zweier unterschiedlicher Strukturbeschreibungen nicht erfaßt, ebensowenig wie die Rekursivität von Sprache(n) beispielsweise was die Verschachtelungstiefe von Konstruktionen angeht. Um diesen Punkt zu ändern, müssen wir einen völlig anderen Aufbau der Grammatik verwenden.

6.1.3. Die Erweiterung der Grammatik um sekundäre grammatische Merkmale[26]

Das Problem bei der Sache ist, dass weder im Lexikon noch in den Satzmustern berücksichtigt ist, ob die kombinierten Elemente bezüglich solcher Merkmale wie Numerus, Person oder Kasus kongruieren. Das heißt, dass Kombinationen wie

**these book* (Problem bzgl. Numerus)

**I kicks you* (Problem bzgl. Person)

**She loves he* (Problem bzgl. Kasus)

allesamt als o.k. angezeigt würden, ohne es zu sein. Wie kann man diesen Zustand revidieren? Nun, wir müssen sowohl das Lexikon als auch die Satzmuster modifizieren dahingehend, dass in ihnen den jeweils vorliegenden Merkmalen bzgl. Numerus, Person usw. Rechnung getragen wird.

Bevor wir uns der Umsetzung in Prolog widmen, müssen wir aber natürlich Klarheit haben, für welche Lexemklasse überhaupt welche Merkmale für unsere Grammatik relevant sind. Informell sähe das – für das Englische! – z.B. so aus:

DETERMINATOREN: Numerus, Definitheit

NOMINA: Numerus, Person, Kasus

PRONOMINA: Numerus, Person, Kasus, Genus

VERBEN: Numerus, Person, Tempus, Modus

PRÄPOSITIONEN, ADJEKTIVE und ADVERBIEN: Keine.

Wir wollen uns für den Einstieg allerdings exklusiv auf das Numerusmerkmal konzentrieren.

Die Erweiterung des Lexikons um ein solches Numerusmerkmal kann **exemplarisch** wie folgt dargestellt werden:

Wort	*Lexemklasse*	*Merkmal*
a	det	singular
these	det	plural
the	det	singular/plural
boy	noun	singular
boys	noun	singular
girl	noun	singular
girls	nomen	plural
kisses	vt	singular
kissed	vt	singular/plural
in	prep	—
ugly	adj	—
very	adv	—

Was die Umsetzung dieser Information im Pattern-Matcher angeht, stellen sich (mindestens) zwei Fragen:

1. Was macht man in den Fällen, in denen ein Wort bezüglich des Merkmales nicht spezifiziert ist, beispielsweise bei *the* und *kissed*?
2. Wie kann man dieses Merkmal überhaupt in das Prolog-Lexikon integrieren?

Die erste Frage ist leicht zu beantworten – in solchen, nicht-festgelegten Fällen verwenden wir einfach die anonyme Variable, sprich den Unterstrich '_'.

[26] Anomale Konstruktionen (?*the bed kicked the garden*) werden nicht berücksichtigt. Prinzipiell könnte deren Behandlung aber nach demselben Prinzip ablaufen, wie es in diesem Abschnitt für die ungrammatischen Konstruktionen vorgestellt wird - sprich über Merkmale wie z.B. [±BELEBT]

Bezüglich Frage 2 könnte man auf die Idee kommen, die lex/2-Fakten einfach um das relevante Merkmale zu erweitern, d.h. dass wir das Merkmal einfach als zusätzliches Argument mitführen, beispielsweise so:

```
lex(a, det, sg).
lex(these, det, pl.)
lex(the,det,_).
lex(boy,noun, sg).
lex(boys,noun, pl).
lex(kisses, vt, sg).
lex(kissed, vt, _).
lex(in, prep).
lex(ugly, adj).
```

So aber funktioniert das nicht. Warum? Die Merkmale eines jeden Eintrages sind zwar auf diese Art im Lexikon erfaßt aber – und das ist das Problem – wir haben nun kein lex/2–Prädikat mehr: wir hätten nämlich für die Determinatoren, Nomina und Verben ein lex/3-Prädikat, für Präpositionen, Adjektive und Adverbien dageben ein lex/2-Prädikat. Diese Situation verschärft sich natürlich dann, wenn es nicht mehr ausschließlich um Numerus, sondern auch noch um andere sekundäre grammatische Kategorien geht.

Wenn wir so vorgingen, würden nämlich weder die Abbildung einer Wortliste auf eine Liste der lexikalischen Kategorien noch der Mustervergleich mehr funktionieren - ganz einfach deshalb, weil wir hierfür Lexikoneinträge brauchen, die **alle** dieselbe Stelligkeit haben. Angesichts der Tatsache, dass die Anzahl der relevanten Merkmale aber für die unterschiedlichen Lexemklassen variiert, scheint das nicht zu funktionieren – es sei denn, dass man die Merkmale nicht als 'eigenständige' Argumente des lex-Funktors notiert, sondern sie stattdessen als Argumente der Kategorienbezeichnung aufführt, denn auf diese Art würde sich an der Stelligkeit von lex/2 nichts ändern. Abstrahiert kann diese Vorgehensweise so notiert werden: LEX(WORT, KAT(MERKMAL$_1$, MERKMAL$_2$, ..., MERKMAL$_N$)). Für die einzelnen Kategorien heißt das:

det(Numerus)
noun(Numerus)
verb(Numerus)
prep
adj
adv

Konkret würden die oa. Lexikoneintrage also die folgende Form haben:

```
lex(a, det(sg)).
lex(these, det(pl)).
lex(boy,noun(sg)).
lex(boys,noun(pl)).
lex(kisses, vt(sg)).
lex(kissed, vt(_)).
lex(in, prep).
lex(ugly, adj).
```

Man sieht, dass auf diese Weise alle Lexikoneinträge die Stelligkeit 2 haben. Die einzelnen Kategorien, die ja nun in Form von Funktor-Argument-Strukturen notiert sind, variieren dagegen in ihrer Stelligkeit - det/1, noun/1 und verb/1 gegenüber prep/0 und adj/0. Soviel zunächst zum Lexikon.

Natürlich müssen aber auch die Satzmuster modifiziert werden, denn hier müssen ja nun Angaben gemacht werden, aus denen das Programm entnehmen kann, dass beispielsweise Determinator und Nomen kongruieren müssen mit Bezug auf das Numerusmerkmal.

Diese Modifikation ist aber einfach. Erst einmal muss berücksichtigt werden, dass die Form der Liste lexikalischer Kategorien ja jetzt anders aussieht. Wenn eine Eingabewortliste auf eine Ausgabeliste lexikalischer Kategorien abgebildet würde, hätte das Ergebnis die folgende Form:

Eingabeliste: [a,boy,sleeps].

Ausgabeliste: [det(sg), noun(sg), vi(sg)].

Hieran sieht man vielleicht schon, wie der Hase laufen muss. Offensichtlich ist es so, das dass einzige Argument von det/1 übereinstimmen muss mit dem einzigen Argument von noun/1. Das einzige Argument von noun/1 muss übereinstimmen mit dem einzigen Argument von vi/2.[27]

Betrachten wir auf diesem Hintergrund das folgende althergebrachte Pattern:

muster([det, noun, vi]).

Dieses kann nun einfach modifiziert werden, indem den Kategorienbezeichnungen die entsprechenden Argumente hinzugefügt werden. In einem Prolog-Muster müssten identische Variablen für den kongruierenden Wert verwendet werden:

muster([det(**N**), noun(**N**), vi(**N**)].

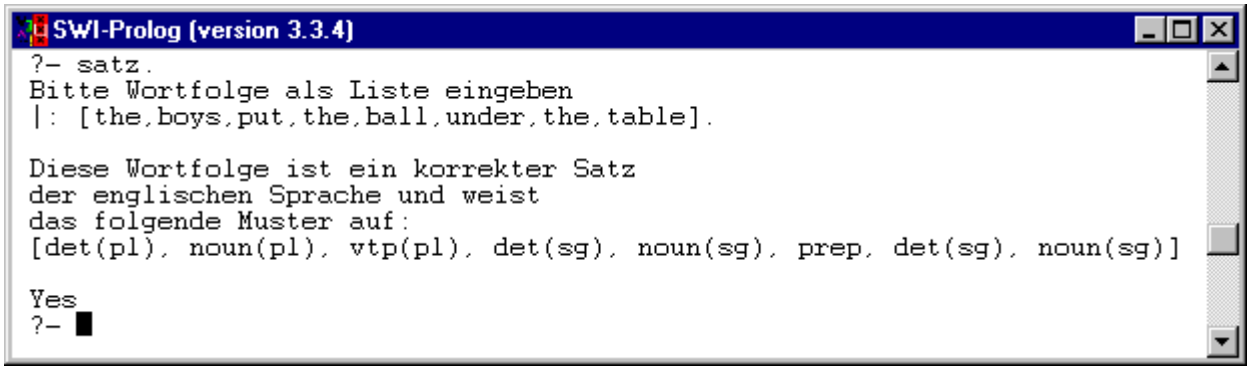
Wenn ein Verb ein Objekt hat, muss dieses bzgl. Numerus natürlich nicht mit dem Verb kongruieren:

*The boy reads **the book** / The boy reads **the books**.*

Ergo muss für den Numeruswert eines Objekts-Nomens (und dessen Determinator) in einer Konstruktion mit einem transitiven Verb eine **andere** Variable vewendet werden:

muster([det(**N**), noun(**N**), vt(**N**), det(**N1**), noun(**N1**)].

Wenn diese Modifikationen durchgeführt werden, würde der Pattern-Matcher nunmehr so aussehen:



Interessant an dieser Stelle ist die Tatsache, dass für einen Det wie *the*, der im Lexikon bezüglich des Numerusmerkmales unspezifiziert ist, hier doch konkrete Werte (im vorliegen Fall PLURAL für *the boys*; SINGULAR für *the ball* und *the table*) ausgegeben wird. In der Veranstaltung werden wir im Zusammenhang mit Unifikationsprozessen noch darüber unterhalten.

Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt **Aufgabe 2** am Kapitelende angehen.

6.1.4. Die Modifikation des Lexikons während der Laufzeit des Programmes

Das zweite der weiter oben angesprochenen Probleme ist die Tatsache, dass natürlich nur diejenigen Wortketten behandelt werden können, für deren Elemente sich entsprechende Einträge im Lexikon finden. Falls der Benutzer eine Kette wie z.B. *the cat purrs* eingibt, die ja ohne Zweifel ok ist, von der aber nicht alle Elemente im Lexikon auftreten, würde auch hier das befürchtete 'No' als Analyseergebnis erscheinen.

Was wir benötigen, ist ein Prädikat, welches ein unbekanntes Wort in der Wort-Eingabeliste erkennt, den Benutzer auffordert, die Kategorie und ggf. das Numerusmerkmal dieses Wortes einzugeben um es dann in der Wissensbasis zur Verfügung zu haben. Es gäbe auch die Möglichkeit, diese neue Information

[27] Auf dem 'einzigen' wird hier ein bißchen rumgeritten deshalb, weil in einer Erweiterung um Person, Tempus etc. natürlich mehr als nur ein Argument in der Struktur auftreten wird.

tatsächlich in die Datei lexikon.pl einschreiben zu lassen – angesichts der Tatsache aber, dass viele Benutzer häufig nur eine etwas vage Vorstellung über die Zugehörigkeit eines Elementes zu einer Lexemklasse haben (jawohl: auch Linguistikstudierende), riskieren wir diese Option lieber nicht.

Es war eigentlich so geplant, dass wir dieses Prädikat (passenderweise nachschlagen/2 genannt) gemeinsam definieren, da es relativ leicht zu durchschauen ist und man dabei etwas über Programmablauf und Datenbankprädikate lernt. Durch die Integration des Numerusmerkmales für bestimmte Kategorien aber hat sich dieses Prädikat verkompliziert, also wird es hier einfach in seinen Grundzügen beschrieben, anschließend komplett in Prolog wiedergegeben und kann dann in das Lexikon eingebaut werden.

Wird nachschlagen(Wort,Katmerk) mit einem Wort aufgerufen, welches **nicht** im Lexikon steht, so

1. informiere den Benutzer über diese Tatsache und bitte ihn, Kat einzugeben,
2. lese Kat
3. Falls Kat *Nomen*, *Verb* oder *Det* ist:
   Erfrage das Numerusmerkmal Num mit hole_merkmal/2.
   Erzeuge aus der Kategorie und dem Numerusmerkmal einen Eintrag Katmerk in Funktor-Argument-struktur
   Falls die Kategorie *Präposition*, *Adjektiv* oder *Adverb* ist:
   Setze Katmerk mit Kat gleich
4. Setze die Variable Eintrag mit einem Fakt lex(Wort, Katmerk) gleich,
5. Frage, ob Eintrag in die Datenbank übernommen werden soll (ja oder nein),
6. wenn die Antwort 'ja' lautet: 'asserte' Eintrag,
7. wenn die Antwort 'nein' lautet, dann true (der Grund dafür wird später klarer (vielleicht)).

```
hole_merkmal(_Kat,Merkmal):-
        write('Bitte das Numerusmerkmal
        eingeben (sg/pl/ _)'),
        nl,
        read(Merkmal).


nachschlagen(Wort,Kat):-
        lex(Wort,Kat),!.
```

```
nachschlagen(Wort,Katmerk):-
        write('Das Wort  '),
        write(Wort),
        write('  steht nicht im Lexikon'),
        nl,
        write('Bitte die Kategorie angeben: '),
        nl,
        read(Kat),
        (
         (member(Kat,[noun,verb,det]),
         hole_merkmal(Kat,Merkmal),
         Katmerk =.. [Kat,Merkmal],!)
         ;
         Katmerk = Kat
        ),
        Eintrag = lex(Wort,Katmerk),
        write('Soll der folgende Eintrag
        in die Datenbank übernommen werden? (ja/nein)'),
        nl,
        write(Eintrag),
        nl,
        read(Antwort),
        nl,
        (Antwort = ja, assert(Eintrag),!;
         Antwort = nein, true, !).
```

Wenn nun die rekursive Regel des Prädikates lexkat/2 entsprechend modifiziert wird:

```
lexkat([Wort|Wortliste],[Kat|Katliste]):-
        nachschlagen(Wort,Kat),
        lexkat(Wortliste,Katliste).
```

werden während der Laufzeit des Programmes neue lex/2 Fakten in die Datenbank übernommen.

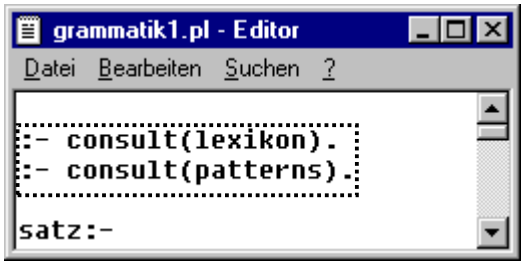
Wenn Ihr das Skript am PC durcharbeitet, könnt Ihr jetzt **Aufgabe 3** am Kapitelende angehen.

Aufgabe 1:

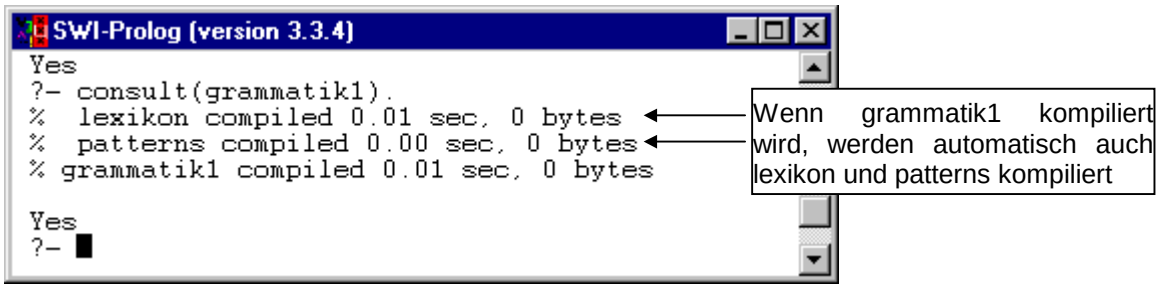
1. Legt eine Datei lexikon.pl an, in der Lexikon von Seite 58 in der Form von lex/2–Fakten realisiert ist.
2. Legt eine Datei patterns.pl an, in der die Satzmuster von Seite 58 in der Form von muster/1–Fakten realisiert sind.
3. Legt eine Datei grammatik1.pl an

   a) Lasst darin die Dateien lexikon.pl und patterns.pl per consult/1 aufrufen.

   Anmerkung:
   Die Datei mit der Grammatik benötigt sowohl die Information aus dem Lexikon als auch die Satzmuster. In diesem Fall bietet es sich an, die entsprechenden Dateien, also lexikon und patterns aus der anderen Datei heraus aufzurufen. Dazu verwendet man – wie auf der Prolog-Oberfläche – consult/1, und zwar als als Befehl in der Datei grammatik1.pl. Dabei ist die Eingabe zu beachten: ein solcher Befehl wird durch das 'Falls'-Zeichen ausgelöst, also **:– consult(lexikon)**, wie in dem nachstehenden Screen-Shot:



   Wenn eine solche Datei eingelesen wird, werden auch automatisch die aus der Datei heraus aufgerufenen Dateien eingelesen:



   b) Definiert anschließend das Prädikat satz/0, welches die in Abschnitt **6.1.2.3** vorgestellten Eigenschaften hat

Aufgabe 2:

1. Erweitert die Datei lexikon.pl (an den relevanten Stellen) um das Numerusmerkmal.
2. Erweitert die Satzmuster in der Datei patterns.pl um das Numerusmerkmal.

Aufgabe 3:

1. Erweitert die Datei lexikon.pl um die Prädikate nachschlagen/2 und hole_merkmal/2
2. Erweitert das Prädikat lexkat/2 in der Datei grammatik1.pl so, dass bei der Abbildung einer Wortliste auf eine Kategorienliste zunächst überprüft wird, ob das Wort im Lexikon steht (setzt hier also nachschlagen/2 ein).

6.1.5. Der Pattern-Matchern als Ersetzungsgrammatik-Akzeptor (d'oh)

Auf S. 60 wurde bereits ausgesagt, dass das zentrale Problem des Pattern-Matchers darin liegt, dass sich die Muster exklusiv auf die lineare Anordnung von (Elementen von) Lexemklassen beziehen und diese durchgehend handgestrickt sind. Was daran verkehrt ist? Nun, die Verwendung von Lexemklassen statt konkreter Wörter bzw. Wortformen stellt wohl eine gewisse Generalisierung dar, besonders weit aber kommt man damit nicht. So kann beispielsweise das rekursive Potential von Sprache (wie es sich ua. in der folgenden Konstruktion manifestiert: *a tree in a garden near a house beside a lake*) in der Grammatik nicht abgebildet werden. Außerdem wird im Pattern-Matcher überhaupt nicht deutlich, dass z.B. die folgenden Konstruktionen strukturell betrachtet große Analogien aufweisen:

I know that { *the cat is hungry* / *the black cat is hungry* / *the black cat is very hungry* }

Hier haben wir es jeweils mit einem eingebetteten Satz zu tun, da uns aber im Pattern-Matcher das Konstrukt 'Satz' genau so wenig zur Verfügung steht wie die Konstrukte NP, VP usw., hätte jede der oa. Formen ein eigenes Muster. Das zentrale Anliegen dieses Abschnittes ist, zu zeigen, wie der Pattern-Matcher um syntaktische Kategorien, sprich Konstituentenklassen, erweitert werden kann dahingehend, dass sich die Muster aus den Regeln für diese syntaktischen Kategorien ergeben.

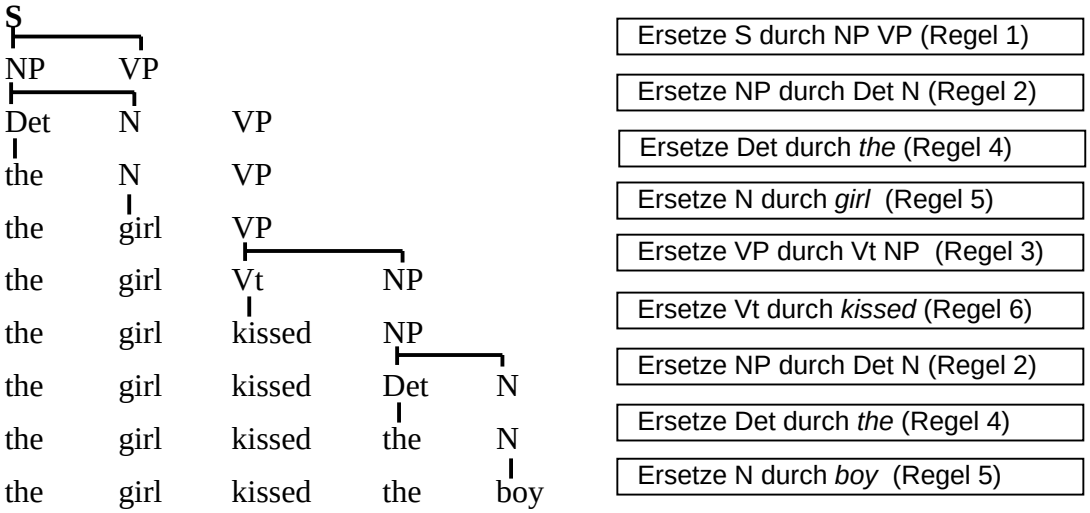
6.1.5.1. Phrasenstrukturregeln als Ersetzungsregeln

Eine der interessanten Fragestellungen bei der Beurteilung einer (modernen, formalen) Grammatik ist die Frage danach, wie man eigentlich von der Grammatik (die sich z.B. aus einem Lexikon und einer Regelkomponente konstituiert) **genau** zu der Aussage kommt, ein Satz sei – mit Bezug auf die Grammatik – wohlgeformt oder nicht.[28] Eine Möglichkeit dafür, diese Ableitung transparent zu machen, besteht darin, Phrasenstruktregeln als Ersetzungsregeln zu interpretieren. Eine Regel wie NP → Det N ist dann eine Aussage, die besagt, dass in einer Symbolkette überall dort, wo das Symbol NP auftritt, dieses durch die Symbolfolge Det–N substituiert werden kann. Betrachten wir dazu die folgende Grammatik:

PS-Regeln	**Lexikon**
1. S → NP VP	4. Det → *the*
2. NP → Det N	5. N → *boy, girl*
3. VP → Vt NP	6. Vt → *kissed*

Ausgehend von dem Startsymbol 'S' kann die Kette *the girl kissed the boy* über die folgende Ableitung als wohlgeformt (mit Bezug auf die Grammatik) identifiziert werden. Dabei ist zu berücksichtigen, dass

1. die Kette von links nach rechts abgearbeitet und
2. pro Ableitungsschritt nur ein Symbol ersetzt wird:



[28] Auf diesen interessanten Punkt werden später im Zusammenhang mit der Frage 'wo kommen die Strukturbeschreibungen eigentlich **genau** her' noch detaillierter eingehen.

6.1.5.2. Die Umsetzung in Prolog

Das Ganze wird hier deshalb so explizit aufgeführt, weil sich für uns natürlich die Frage stellt, wie man so etwas in Prolog umsetzen kann. Entgegen der gängigen Praxis in diesem Skript wird dieser Punkt in den nächsten Abschnitten relativ straight-forward angegangen, denn eine umgangsprachliche Formulierung der Aufgabenstellung würde wahrscheinlich mehr verwirren als erklären.

Wir schreiben ein Prädikat match/2, welches eine Wortliste auf eine 'Ableitungsliste' abbildet und gehen von der folgenden Anfrage aus:

?- match([the,boy,sleeps],[s]).

Umgangssprachlich: kann die Liste [the,boy,sleeps] auf die Liste [s] abgebildet werden; noch umgangssprachlicher: ist die Kette *the boy sleeps* ein Satz? Bei dieser Anfrage passiert folgendes:

Wortliste	Ableitungsliste	Aktion	Wissensbasis
[the,boy,sleeps]	**[s]**		
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungliste einen Lexikoneintrag bildet:	FAIL! (Keine Regel s→ the)
		Wenn nicht: suche nach einer PS-Regel, in der das erste Element der Ableitungsliste auf der linken Seite auftritt:	OK! (Regel s → np vp)
		Ersetze das Symbol in der Ableitungskette durch das/die Symbol/e auf der rechten Seite der PS-Regel. Das Ergebnis ist ein neues Listenpaar.	
[the,boy,sleeps]	**[np,vp]**	←	
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungsliste einen Lexikoneintrag bildet:	FAIL! (Keine Regel np→ the)
		Wenn nicht: suche nach einer PS-Regel, in der das erste Element der Ableitungsliste auf der linken Seite auftritt:	OK! (np → det noun)
		Ersetze das Symbol in der Ableitungsliste durch das/die Symbol/e auf der rechten Seite der PS-Regel. Das Ergebnis ist ein neues Listenpaar.	
[the,boy,sleeps]	**[det,noun,vp]**	←	
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungsliste einen Lexikoneintrag bildet:	OK! (det → the)
		Schneide in beiden Listen den Kopf ab und mache mit den Listenresten weiter. Das Ergebnis ist ein neues Listenpaar.	
[boy,sleeps]	**[noun,vp]**	←	
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungliste einen Lexikoneintrag bildet:	OK! (noun → boy)
		Schneide in beiden Listen den Kopf ab und mache mit den Listenresten weiter. Das Ergebnis ist ein neues Listenpaar.	
[sleeps]	**[vp]**	←	
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungliste einen Lexikoneintrag bildet:	FAIL! (Keine Regel vp → sleeps)
		Wenn nicht: suche nach einer PS-Regel, in der das erste Element der Ableitungsliste auf der linken Seite auftritt:	OK! (vp → vi)
		Ersetze das Symbol in der Ableitungsliste durch das/die Symbol/e auf der rechten Seite der PS-Regel. Das Ergebnis ist ein neues Listenpaar.	
[sleeps]	**[vi]**	←	
	→	Prüfe, ob das erste Element der Wortkette mit dem Kopf der Ableitungliste einen Lexikoneintrag bildet.	OK! (vi → sleeps)
		Schneide in beiden Listen den Kopf ab und mache mit den Listenresten weiter. Das Ergebnis ist ein neues Listenpaar:	
[ ]	**[ ]**	←	
	→	Wenn die Listen leer sind: **TRUE** (Beende die Prozedur)	
Ergebnis: **YES**			

Wie die Aktionsspalte zeigt, müssen bei der Abarbeitung von Wort- und Ableitungsliste zwei Fälle klar voneinander unterschieden werden:

- der Kopf der Wortliste und der Kopf der Ableitungsliste sind über einen Lexikoneintrag zueinander in Beziehung gesetzt – dieser Fall tritt logischerweise erst dann ein, wenn der Kopf der Ableitungsliste eine lexikalische, keine syntaktische Kategorie ist
- der Kopf der Wortliste und der Kopf der Ableitungsliste sind nicht über einen Lexikoneintrag zueinander in Beziehung gesetzt – dieser Fall ist immer dann gegeben, wenn es sich bei dem Kopf der Ableitungsliste um eine syntaktische Kategorie handelt.

Es geht letzendlich um nichts anderes, als – ausgehend von einem Startsymbol 'S' – in der Ableitungsliste schrittweise die Ersetzungen vorzunehmen, die in den PS-Regeln vorgegeben sind, und zwar solange, bis die Ebene der lexikalischen Kategorien erreicht ist. Wenn wir uns die Spalte der Ableitungsliste ansehen, stellen wir - beispielsweise beim Übergang der Liste [np, vp] zur Liste [det, noun, vp] einen wichtigen Aspekt dieser Ersetzungen fest, nämlich die Tatsache, dass ja immer nur ein Symbol zur Zeit substituiert wird, nämlich der **Kopf** der Ableitungsliste (remember: die Abarbeitung erfolgt von links nach rechts), hier also das Symbol 'np' durch die Symbolfolge 'det noun', während der **Rest** der Ableitungsliste (hier: das Symbol 'vp') in die neue Ableitungsliste übernommen werden muss. Die neue Ableitungsliste ist mithin die Kombination aus

1. dem Symbol / der Symbolfolge, durch die der Kopf der alten Ableitungsliste laut einer PS-Regel substituiert wird
2. dem Rest der alten Ableitungsliste

Das müssen wir für die Umsetzung unbedingt im Hinterkopf behalten!

Kommen wir jetzt zur Realisierung in Prolog. Unser Programm wird erneut drei Komponenten umfassen, nämlich ein Lexikon, eine Reihe von PS-Regeln (anstatt der handgestrickten Muster) und die Steuerdatei, die die Abarbeitung einer vom Benutzer eingegebenen Wortliste regelt.

6.1.5.3. Das Lexikon

Das Lexikon wird in Form der altbekannten lex/2-Fakten notiert. Wir verzichten zunächst auf Fisimatenten wie Flexionsmerkmale, kommen darauf aber wieder zurück. (Für diejenigen, die das 'alte' Lexikon ohne diese Merkmale nicht in irgendeiner Weise beibehalten haben, bedeutet das leider, das Lexikon nochmal anzulegen).

6.1.5.4. Die PS-Regeln

Wir übersetzen die PS-Regeln in Form von regel/2-Fakten in Prolog. Das erste Argument von regel/2 ist das Symbol zur linken des Pfeils, das als Atom wiedergegeben ist, das zweite Argument ist eine Liste der auf der rechten Seite des Pfeils aufgeführten Konstituenten. Wichtig: auch in denjenigen Fällen, in denen auf der rechten Seite des Pfeiles nur ein einziges Symbol auftritt (z.B. bei VP $\rightarrow V_i$) muss dieses in Listenform notiert werden. Die weiter oben aufgeführten PS-Regeln hätten mithin die folgende Form:

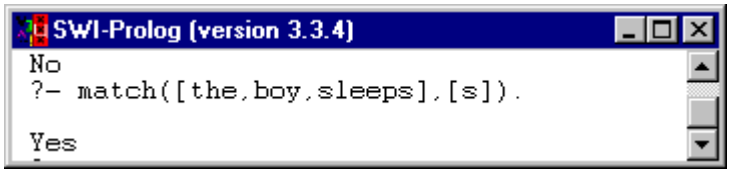
```
regel(s, [np, vp]).
regel(np, [det, noun]).
regel(vp, [vi]).
```

6.1.5.5. Die Steuerdatei

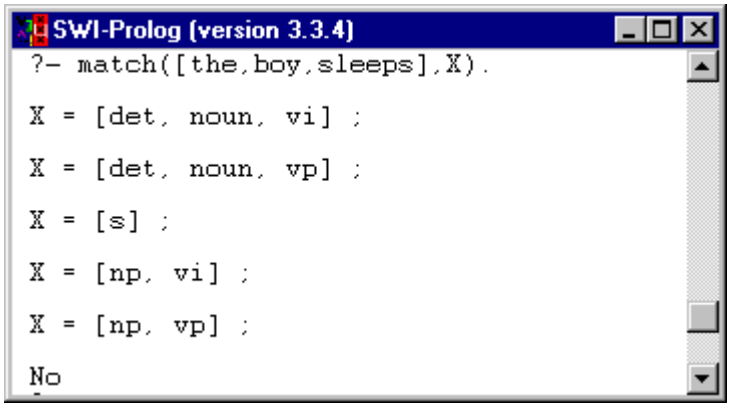
Hier liegt der Hase im Pfeffer. Wir wollen ein Prädikat match/2 schreiben. Das erste Argument ist die zu analysierende Wortkette; das zweite Argument ist die Ableitungsliste:

match(Wortliste,Ableitungsliste).

Eine Anfrage mit instantiiertem zweiten Argument (dem Startsymbol 'S') soll als Ergebnis die Meldung **Yes** bekommen:



In den Fällen, in denen das zweite Argument ein Variable ist, können die einzelnen Ableitungsschritte angezeigt werden (die Reihenfolge ist hier irrelevant):



Das Prädikat match/2 ist in drei Regeln notiert:

1. die Abbruchbedingung
2. der Fall, in denen match/2 auf das Lexikon rekurriert (rekursive Regel)
3. der Fall, in denen match/2 auf die PS-Regeln rekurriert (ebenfalls rekursive Regel)

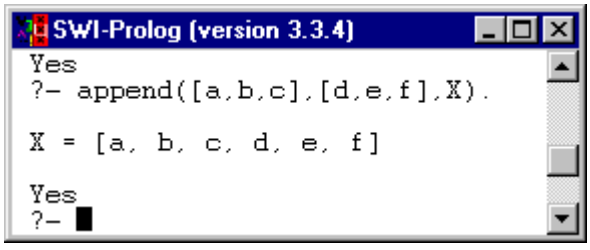
Regel 1:
Die Abbruchbdingung ist klar:
match([ ], [ ]).

Regel 2 (Bezug zu lex/2):
In der zweiten Regel wird auf das Lexikon Bezug genommen:

```
match([Wort|Restwortliste], [Kategorie|Restkategorienliste]):-
        lex(Wort,Kategorie),
        match(Restwortliste,Restkategorienliste).
```

Sobald es der Fall ist, dass eine Abbildung zwischen einem Wort und einer lexikalischen Kategorie erfolgt, werden im rekursiven Aufruf der Regel die Köpfe der Listen abgeschnitten und das Prädikat mit den Listenresten erneut aufgerufen.

Regel 3 (Bezug zu regel/2):
Durch die Formulierung der zweiten Regel mit Bezug auf die lex/2-Fakten ist für die dritte Regel von vorneherein klar, dass sie nur zur Anwendung kommt, wenn das erste Symbol der Ableitungsliste (sprich der Kopf der Ableitungsliste) eine syntaktische Kategorie ist. Dabei ist zu berücksichtigen, dass sich durch die Anwendung der dritten Regel an der Wortliste selber nichts ändert. Dazu braucht man sich bloß die Tabelle auf Seite 67 anzusehen – wann immer eine Ersetzung über eine PS-Regel, also nicht über eine Lexikonregel erfolgt, bleibt die Wortliste in voller Form erhalten. Prolog-technisch betrachtet hat das die Konsequenz, dass das erste Argument von match/2 in der dritten Regel beim rekursiven Aufruf nicht angetastet wird, ergo muss hier auch keine Zerlegung in einen Kopf und einen Rest erfolgen. Das zweite Argument hingegen muss in einen Kopf und einen Rest zerlegt werden. Der Kopf – eine syntaktische Kategorie – wird dann in den regel/2-Fakten 'gesucht'. Wird dort eine Regel gefunden, muss folgendes passieren: das zweite Argument der Regel (welches ja in Form einer Liste notiert ist) wird mit dem Rest der Ableitungsliste zu der 'neuen' Ableitungsliste verbunden. Dafür benutzen wir ein eingebautes Prolog-Prädikat namens append/3, welches die Funktion hat, zwei Listen zu einer dritten Liste zusammen-zuhängen und wie folgt aussieht:

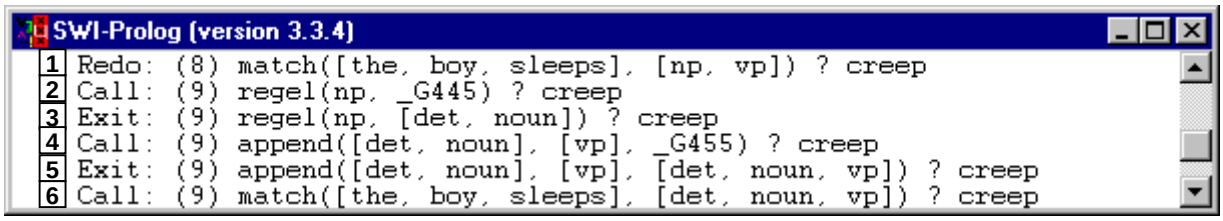


Das dritte Argument von append/3 ist eine neue Liste, die aus der Verkettung des ersten und des zweiten Argumentes (jeweils Listen) besteht, und genau diese Funktion benötigen wir.

Also kann die dritte Regel wie folgt notiert werden:

```
match(Wortliste, [Kategorie|Restkategorienliste]):-
        regel(Kategorie,Konstituenten),
        append(Konstituenten,Restkategorienliste,Neue_Ableitungsliste),
        match(Wortliste,Neue_Ableitungsliste).
```

Der nachfolgende kommentierte Screen-Shot gibt einen Teil der getracten Abarbeitung wieder, der genau zeigt, wie das Ganze funktioniert. Die Zeilennumerierung ist nachträglich hinzugefügt:



In Zeile 1 wird match/2 mit der Wortliste [the, boy, sleeps] und der Ableitungsliste [np,vp] aufgerufen. In Zeile 2 wird überprüft, ob es ein regel/2 Fakt gibt, in dem der Kopf der Ableitungsliste, also np, das erste Argument ist. Zeile 3 zeigt das Ergebnis an - es gibt eine entsprechende Regel, und die Konstituenten, durch die np substituiert werden soll, sind in Form der Liste [det, n] aufgeführt. In Zeile 4 nun wird diese Liste per append/3 mit dem Rest der Ableitungsliste – sprich der Liste [vp] zu einer neuen Liste verkettet, das Ergebnis ist in Zeile 5 zu sehen: die Liste [det, noun, vp]. Mit dieser neuen Ableitungsliste wird match/2 (mit unverändertem ersten Argument!) in Zeile 6 rekursiv aufgerufen.

Wie man sieht, ist match/2 eigentlich furchtbar einfach und umfasst insgesamt nur 8 Programmzeilen – deren Erklärung aber erfordert ein ziemlich langes und umständliches Geschreibe. Wie bereits häufiger in diesem Skript wird deshalb einerseits empfohlen, die Sache (der nachstehenden Aufgabe entsprechend) nachzuprogrammieren und dann üppigen Gebrauch von der trace-Funktion zu machen, die häufig mehr sagt als tausend Worte, und andererseits auf die animierte Darstellung in der Sitzung verwiesen, die möglicherweise auch zum besseren Verständnis beiträgt.

Aufgabe 4:

4. Legt eine Datei psg_lex.pl an, in der Lexikon von Seite 58 in der Form von lex/2–Fakten realisiert ist (also ohne Flexionsmerkmale)
5. Legt eine Datei psg_reg.pl an, in der die den Sätzen auf Seite 58 zugrundeliegenden PS-Regeln in der Form von regel/2–Fakten realisiert sind.
6. Legt eine Datei namens psg_pat.pl an
   a) Lasst darin die Dateien psglex.pl und psg_reg.pl per consult/1 aufrufen.
   b) Definiert darin das Prädikat match/2 wie in diesem Kapitel vorgestellt. Macht Euch innerlich dafür bereit, zu diesem Prädikat noch eine Reihe unangenehmer Fragen zu beantworten (schreibt es also nicht einfach ab, sondern versucht, es wirklich zu verstehen).

Im nächsten Kapitel werden wir einen anderen Zugang zur Implementierung einer Phrasenstrukturgrammatik wählen. Von seiner inneren Ausstattung her ist das hier vorgestellte Programm bereits schon jetzt in der Lage, zahlreiche potentielle Modifikationen (von Erweiterungen in Richtung Kongruenz- und Rektionsphänomenen über Ambiguitäten und Verschachtelungen bis zur X-bar Syntax) zu durchlaufen. Das Problem liegt in der Erzeugung einer Strukturbeschreibung. Das soll nicht heißen, dass das nicht ginge - es ist aber ein wenig komplizierter, als es zum jetzigen Zeitpunkt sinnvoll wäre.

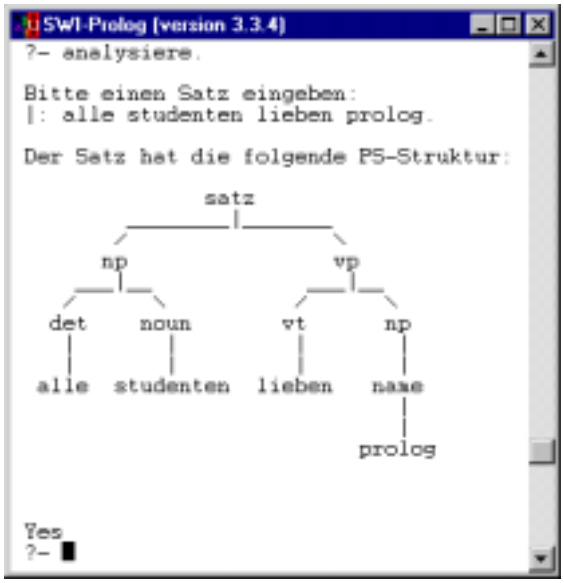
Kapitel 7.
Eine Phrasenstrukturgrammatik mit Prolog

Wie am Ende des vorherigen Kapitels bereits gesagt wurde, ist der um die Ersetzungs-PS-Regeln erweiterte Pattern-Matcher-Akzeptor zweifellos in der Lage, bezüglich derjenigen Komponenten modifiziert zu werden, die das Ganze aus linguistischer Perspektive interessant machen. Das Problem besteht dabei im Aufbau des Programmes: die Erweiterung dahingehend, dass Sätze nicht nur akzeptiert werden, sondern für einen jeden Satz auch eine bzw. mehrere Strukturbeschreibungen generiert werden, ist zwar machbar, aber ein wenig kniffelig. Diese Komplikation umgehen wir, indem wir die Implementierung der Phrasenstruktur von vorneherein anders anlegen als den Patter-Matcher.

In den nächsten Abschnitten wird es darum gehen, schrittweise ein Programm zu entwickeln, welches die folgenden Leistungen aufweist:

1. Es kann entscheiden, ob ein englischer Satz syntaktisch wohlgeformt ist
2. Es kann einem wohlgeformten Satz eine entsprechende Strukturbeschreibung zuordnen.
3. Bei struktureller Ambiguität eines Satzes wird für jede Interpretation eine Strukturbeschreibung geliefert.

Der nachstehende Screen-Shot zeigt, wie das Ganze letztendlich aussehen soll:



Natürlich kann dieses Programm nicht umfassend und erschöpfend sein dahingehend, dass es für alle erdenkbaren Sätze diese Aufgaben erfüllt. Es soll vielmehr eine erste Vorstellung davon erzeugen, welche Problemstellungen in diesem Bereich der maschinellen Sprachverarbeitung auftreten und wie man bestimmte Erkenntnisse aus dem Bereich der Syntaxtheorie zweckdienlich in einem Computerprogramm umsetzen kann. Das Programm ist für die englische Sprache ausgelegt, weil in dieser bestimmte Unannehmlichkeiten, die eine Analyse erschweren, gar nicht erst auftreten (wie z.B. die Flexion der Artikel und Adjektive oder die diskontinuierlichen Elemente im Verbalsyntagma). Ansatzsweise aber werden – wie im Pattern-Matcher auch – diese Phänomene ebenfalls angegangen.

Wir werden zunächst kleine Brötchen backen und Punkt 1. der oa. Aufgabenstellungen bearbeiten, d.h. einen auf einer Phrasenstrukturgrammatik basierenden Akzeptor realisieren, um das Programm dann sukzessive um die anderen Punkte (zu denen neben der Generierung einer Strukturbeschreibung auch die Art und Weise, in der zu analysierende Wortketten eingegeben werden, gehört) zu erweitern und verbessern.

7.1. Teil 1: Ein PSG-Akzeptor

An dem folgenden Beispielsatz soll die Grundlage für das Analyseprogramm erklärt werden:

7.1. *the dog chased the cat*

Ein wesentlicher Aspekt der menschlichen Sprachverarbeitung (der allerdings so unbewußt abläuft, dass man ihn sich erstmal bewußt machen muß) ist die Tatsache, dass eine solche Wortfolge völlig automatisch linear abgearbeitet wird – was die englische geschriebene Sprache betrifft von links nach rechts. Es ist also ganz klar, dass zuerst das Wort *the* kommt, dass *chased* zwischen *the* und *dog* steht, dass die Wortfolge mit *cat* endet usw. Die Reihenfolge der einzelnen Wörter ist natürlich, wie wir bereits im Abschnitt über den Pattern-Matcher gesehen haben, ein zentrales Kriterium bei der Analyse eines Satzes. Was damit gesagt werden soll ist, dass die lineare Reihenfolge der Wörter wie eben in *the dog chased the cat* eine der implizit im Satz enthaltenen Informationen ist, die für Prolog explizit gemacht werden muß (ähnliche Fragestellungen tauchten ja bereits im Stammbaum und im semantischen Netz auf).

Dazu bietet es sich an, den Satz wie folgt zuerst einmal in einzelne Segmente zu zerteilen:

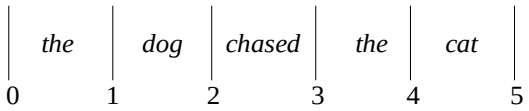


Abb. 7.1

Die Numerierung der einzelnen Positionsstriche ermöglicht, über den Satz *the dog chased the cat* folgende Aussage zu machen:

Der Satz the *dog chased the cat* besteht aus fünf Segmenten, wobei die Anfangsposition 0 mit der Position 1 durch das Wort *the* verbunden ist, die Position 1 mit der Position 2 durch das Wort *dog*, die Position 2 mit der Position 3 durch das Wort *chased*, die Position 3 mit der Position 4 durch das Wort *the* und die Position 4 mit der Endposition 5 durch das Wort *cat*.

Die Kette der einzelnen Wörter als Verbindung jeweils zweier Positionen kann wie folgt graphisch illustriert werden:

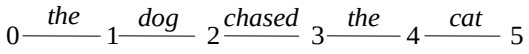


Abb. 7.2.

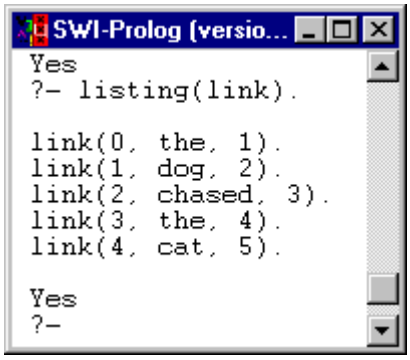
Es stellt sich zunächst die Frage, wie eine solcher Satz in einer für die Weiterverarbeitung durch Prolog geeigneten Form repräsentiert werden kann. Für den Benutzer das einfachste Verfahren wäre es, wie im Pattern-Matcher Sätze als Listen über die Tastatur einzugeben, und den Rest Prolog zu überlassen. Genau so wollen wir verfahren. Im Augenblick aber beschränken wir uns erstmal darauf, diese Information in Form von Fakten einzugeben, und zwar mithilfe eines Prädikats link/3, bei welchem das erste Argument die Ausgangsposition ist, das zweite Argument das verbindende Wort und das dritte Argument die Zielposition. Das Wort *the* stellt die Verbindung her zwischen den Positionen 0 und 1, also gilt als Prolog-Fakt:

```
link(0,the,1).
```

Das Wort *dog* stellt die Verbindung her zwischen den Positionen 1 und 2, ergo

```
link(1,dog,2).
```

und so weiter. Mit den uns bis jetzt zur Verfügung stehenden Mitteln müssen wir erstmal **jeden** Satz, den wir analysieren wollen, auf diese Art eingeben.[29] Dieses Verfahren werden wir weiter unten allerdings automatisieren. Für die Weiterverbeitung (sprich Analyse) muss der Satz aus Beispiel 7.1. wie folgt in der Prolog-Wissensbasis repräsentiert sein:



Hier mag man sich fragen – wieso so umständlich? Wenn wir die zu analysierende Wortkette in Listenform eingeben, also analog zur Herangehensweise im Pattern-Matcher, wird deren Linearität doch viel direkter abgebildet, als es hier der Fall ist. So wir von jedem im zu analysierenden Satz vorkommenden Wort wissen, zu welcher Wortart es gehört (und diese Information ist im Lexikon, welches wir auch hier verwenden, in Form von lex/2-Fakten notiert) sind wir jetzt in der Tat nicht weiter als im Pattern-Matcher (ohne Satzmuster). Es soll an dieser Stelle auch nicht verheimlicht werden, dass das ganze anstehende Verfahren extrem vereinfacht werden könnte, wenn man den in Prolog eingebauten **D**(efinite) - **C**(lause) - **G**(rammar)-Formalismus verwendet. In diesem Formalismus, auf den wir später ggf. näher eingehen werden und der am Ende des Kapitels genauer vorgestellt wird, sind ganz bestimmte Verfahrensschritte bei der Umsetzung einer Phrasenstrukturgrammatik in Prolog bereits

[29]Das bedeutet, dass ein Satz wie *the teacher wrote a book* ebenfalls in 'mundgerechten Prolog–Happen' eingegeben würde:
link(0,the,1). link(3,a,4).
link(1,teacher,2). link(4,book,5).
link(2,wrote,3).

vordefiniert, so dass man sich bei der Implementierung über bestimmte Dinge – wie eben 'in welcher Form muss die Wortkette in der Wissensbasis repräsentiert sein' überhaupt keine Gedanken mehr machen muss. Da es aber ein Ziel der Veranstaltung ist, über die Vermittlung einiger zentraler Prolog-Grundlagen die Teilenehmenden in die Lage zu versetzen, bestimmte Problemstellungen mehr oder weniger 'from scratch' selbständig angehen zu können, gehen wir zunächst den etwas umständlicheren, dafür aber in seinen einzelnen Schritten völlig transparenten Weg. Auf dieser Basis lassen sich dann vordefinierte Programmschemata wie der DCG-Formalismus auch besser nachvollziehen. Damit zurück zum Problem.

Die Kombination der link/3 Fakten und des Lexikons liefert uns sozusagen die lineare Ebene des Satzes. In der PSG geht es aber ja um mehr – nämlich auch um die hierarchische Ebene der syntaktischen Kategorien. An dieser Stelle wird deutlich, wieso wir die link/3-Notation für die zu analysierende Wortkette verwenden: um nämlich die syntaktischen Kategorien für den gegebenen Satz darzustellen, haben wir durch die Einteilung des Satzes in einzelne Segmente bereits eine vorzügliche Basis geschaffen. Die Strukturbeschreibung von *the dog chased the cat* hätte (im Rahmen der traditionellen Konstituentenanalyse) auf dieser Basis die folgende Form:

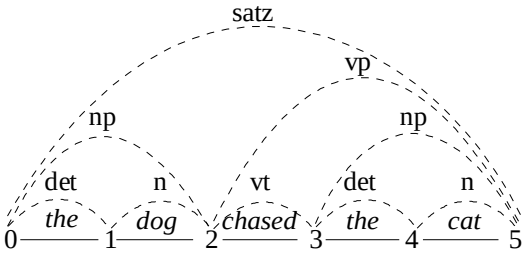


Abb. 7.3.

Nun können die folgenden Aussagen getroffen werden:

1. Der Bogen, der die Positionen 0 – 5 umspannt, entspricht einem Satz. Die Position 2 zerlegt diese Spanne derart, dass der Bogen von 0 – 2 einer Nominalphrase entspricht und der Bogen von Position 2 – 5 einer Verbalphrase.
2. Der Bogen, der die Positionen 0 – 2 umspannt, entspricht einer Nominalphrase. Position 1 zerlegt die Spanne von 0 – 2 derart, dass der Bogen von 0 – 1 einem Determinator entspricht und der Bogen von 1 – 2 einem Nomen.
3. Der Bogen, der die Positionen 2 – 5 umspannt, entspricht einer Verbalphrase. Position 3 zerlegt die Spanne von 2 – 5 derart, dass der Bogen von 2 – 3 einem transitiven Verb entspricht und der Bogen von 3 – 5 einer Nominalphrase.
4. Der Bogen, der die Positionen 3 – 5 umspannt, entspricht einer Nominalphrase. Position 4 zerlegt die Spanne 3 – 5 derart, dass der Bogen von 3 – 4 einem Determinator entspricht und der Bogen von 4 – 5 einem Nomen.

Zunächst betrachten wir die Aussage unter Punkt 1:

> Der Bogen, der die Positionen 0 – 5 umspannt, entspricht einem Satz. Die Position 2 zerlegt diese Spanne derart, dass der Bogen von 0 – 2 einer Nominalphrase entspricht und der Bogen von Position 2 – 5 einer Verbalphrase.

Diese Aussage, die sich auf den konkreten Satz und also die konkreten Positionen 0, 2 und 5 bezieht, kann auf die folgende Art verallgemeinert und paraphrasiert werden:

Ein Bogen, der von einer Position P0 zu einer Position Pn reicht, ist dann ein Satz, wenn zwischen P0 und Pn eine weitere Position P1 liegt, welche die Spanne P0 – Pn so zerlegt, dass P0 – P1 eine Nominalphrase ist und P1 – Pn eine Verbalphrase.

Dieser verallgemeinerte Ausdruck kann direkt in eine Prolog-Regel übersetzt werden:

```
satz(P0,Pn):-
        np(P0,P1),
        vp(P1,Pn).
```

Dabei sind P0, P1, Pn Variablen, die Werte für unterschiedliche Positionen annehmen können. Auf die gleiche Weise verfahren wir mit den anderen Bögen:

> Der Bogen, der die Positionen 0 – 2 umspannt, entspricht einer Nominalphrase. Position 1 zerlegt die Spanne von 0 – 2 derart, das der Bogen von 0 – 1 einem Determinator entspricht und der Bogen von 1 – 2 einem Nomen.

Auch diese Aussage wird verallgemeinert paraphrasiert:

Ein Bogen, der von einer Position P0 zu einer Position Pn reicht, ist dann eine Nominalphrase, falls zwischen P0 und Pn eine weitere Position P1 liegt, welche die Spanne P0 – Pn so zerlegt, dass P0 – P1 ein Determinator ist und P1 – Pn ein Nomen.

Als Regel:

```
np(P0,Pn):-
        det(P0,P1),
        noun(P1,Pn).
```

An dieser Stelle schon sieht man den Vorteil der Verallgemeinerung: mit dieser Regel für die Nominalphrase ist nicht nur der Bogen, der in Abbildung **7.3.** die Positionen 0 – 2 umspannt (*the dog*) erfasst, sondern auch der Bogen, der die Positionen 3 – 5 umspannt (*the cat*).

Nun fehlt nur noch die Regel für den Verbalphrasenbogen:

> Der Bogen, der die Positionen 2 – 5 umfaßt, entspricht einer Verbalphrase. Position 3 zerlegt die Spanne von 2 – 5 derart, dass der Bogen von 2 – 3 einem transitiven Verb entspricht und der Bogen von 3 – 5 einer Nominalphrase.

Verallgemeinert:

Ein Bogen, der von einer Position P0 zu einer Position Pn reicht, ist dann eine Verbalphrase, wenn zwischen P0 und Pn eine weitere Position P1 liegt, welche die Spanne P0 – Pn so zerlegt, dass P0 – P1 ein transitives Verb ist und P1 – Pn eine Nominalphrase.

Als Regel:

```
vp(P0,Pn):-
        vt(P0,P1),
        np(P1,Pn).
```

Die Regeln auf der phrasalen Ebene sind somit für den Satz, die Nominalphrase und die Verbalphrase eingegeben. Was noch fehlt, sind Regeln für die einzelnen lexikalischen Kategorien Determinator, Nomen, und Verb. Dazu betrachten wir die folgenden, wiederum auf Abbildung **7.3.** bezogenen Feststellungen:

1. Die Positionen 0 – 1 werden durch das Wort *the* verbunden, welches der lexikalischen Kategorie DETERMINATOR angehört.
2. Die Positionen 1 – 2 werden durch das Wort *dog* verbunden, welches der lexikalischen Kategorie NOMEN angehört.
3. Die Positionen 2 – 3 werden durch das Wort *chased* verbunden, welches der lexikalischen Kategorie TRANSITIVES VERB angehört.
4. Die Positionen 3 – 4 werden durch das Wort *the* verbunden, welches der lexikalischen Kategorie DETERMINATOR angehört.
5. Die Positionen 4 – 5 werden durch das Wort *cat* verbunden, welches der lexikalischen Kategorie NOMEN angehört.

Auch hier verallgemeinern wir und paraphrasieren die erste Aussage wie folgt:

Ein Bogen, der von einer Position P0 zu einer Position Pn reicht ist dann ein Determinator, wenn P0 und P1 durch ein Wort verbunden sind, welches der lexikalischen Kategorie Determinator angehört.

Diese Aussage wird wie folgt in Prolog übersetzt:

```
det(P0,P1):-
        link(P0,Wort,P1),
        lex(Wort,determinator).
```

Auf das Lexikon bezogen findet sich wg. Schreibfaulheit für die letzte Zielklausel lex(Wort,determinator) wahrscheinlich nur ein einziger Eintrag: Wort = the. Aber dennoch wird auch hier der Vorteil der Generalisierung deutlich, schließlich kann das Lexikon durch eine beliebige Zahl von Einträgen erweitert werden, auf welche die Regel für den Determinator dann angewendet werden kann:

```
lex(a,determinator).
lex(this,determinator).
lex(my,determinator). usw.
```

Nun können wir, analog zur Regel für den Determinator, die Regeln für die anderen lexikalischen Kategorien aufführen:

```
noun(P0,Pn):-
        link(P0,Wort,P1),
        lex(Wort,noun).
vt(P0,Pn):-
        link(P0,Wort,P1),
        lex(Wort,vt).
```

Wenn diese Daten in Prolog eingegeben sind und das Lexikon konsultiert wurde, enthält die Wissensbasis

- die Regeln für den Satz und die phrasalen Kategorien NP und VP
- die Regeln für die lexikalischen Kategorien N, Det und Vt,
- den zu analysierenden Satz *the dog chased the cat* in Form von link/3 Fakten

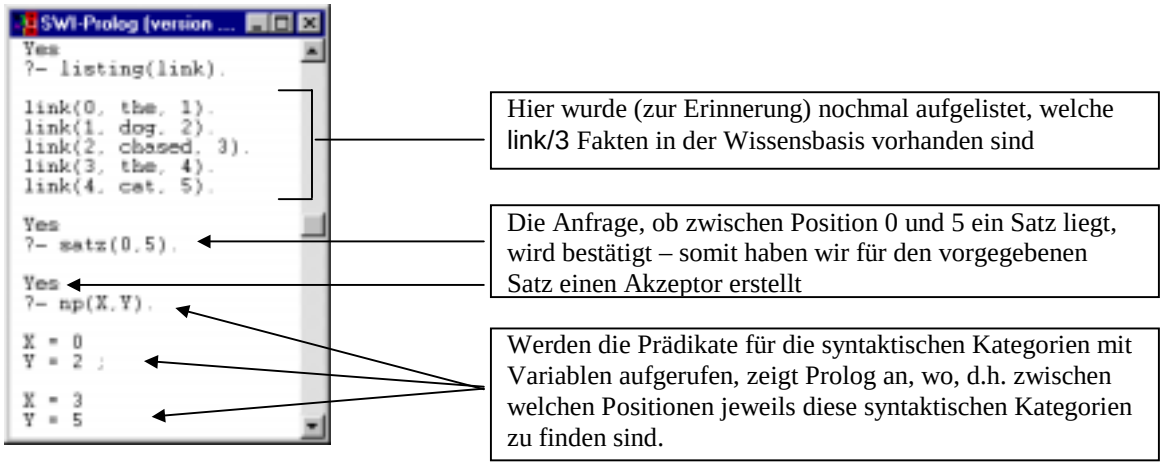
An dieser Stelle können bestimmte Anfragen an das Programm gestellt werden. Beispielsweise können wir überprüfen, ob zwischen den Positionen 0 und 5 ein Satz liegt – ob also die als verbindet/3 Fakten eingegebene Wortkette ein Satz ist:

Liegt zwischen Position 0 und Position 5 ein (wohlgeformter) Satz?
Eingabe: satz(0,5) Ausgabe: yes

Wir können aber, durch die Verwendung von Variablen, auch anzeigen lassen, zwischen jeweils welchen Positionen eine NP, eine VP oder ein Satz liegt:

Zwischen welchen Positionen liegt eine Nominalphrase?
Eingabe: np(X,Y) Ausgabe: X = 0, Y = 2
X = 3, Y = 5

Das Ganze sieht 'in echt' so aus:



Damit wäre das erste Ziel erreicht – wir haben auf der Basis einer PSG einen Aktzeptor geschrieben, der für eine englische Worfolge (in unserem Fall allerdings, aufgrund der schmalen Datenlage, leider nur die

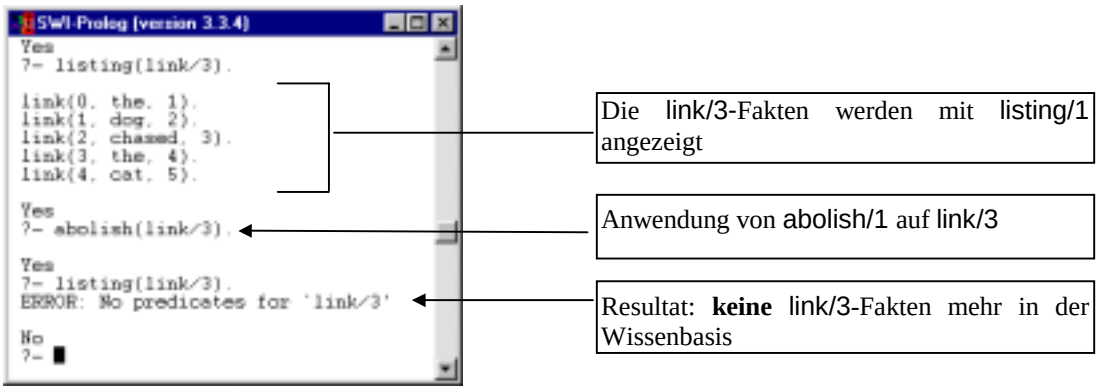
eine Wortfolge und nur für Sätze mit der Struktur ((Det Noun)$_{NP}$ (Vt (Det Noun)$_{NP}$)$_{VP}$)$_{S}$) erkennt, ob diese ein Satz ist oder nicht.

Mit Bezug auf eine PSG mit traditioneller Schreibweise umfaßt die Grammatik die folgenden Regeln:

PSG	Prolog-Regeln	
S → NP VP	s(P0,Pn):- np(P0,P1), vp(P1,Pn).	Regeln für die syntaktischen Kategorien
NP → Det N	np(P0,Pn):- det(P0,P1), noun(P1,Pn).	
VP → Vt NP	vp(P0,Pn):- vt(P0,P1), np(P1,Pn).	
	noun(P0,Pn):- link(P0,Wort,Pn), lex(Wort, noun). det(P0,Pn):- link(P0,Wort,Pn), lex(Wort, det). vt(P0,Pn):- link(P0,Wort,Pn), lex(Wort, vt).	Regeln für die lexikalischen Kategorien
Det → *the*	lex(the,det).	Lexikon
N → {*dog, cat*}	lex(dog,noun). lex(cat,noun).	
Vt→ *chased*	lex(chased,vt).	

Um weitere Sätze zu überprüfen, muss folgendes geschehen:

- Zuerst müssen die alten link/3-Fakten aus der Wissensbasis entfernt werden. Dafür verwendet man das Systemprädikat abolish/1, Das Argument von abolish/1 ist eine Angabe über Funktor und Stelligkeit des zu entfernenden Prädikates. Um also alle link/3-Fakten aus der Wissensbasis zu entfernen, muss folgender Befehl eingegeben werden: abolish(link/3)



- Der neue Satz muss in Form von link/3 Fakten eingegeben werden

In den nachfolgenden Abschnitten werden wir dieses Programm so erweitern, dass eine Strukturbeschreibung erzeugt wird, und die Benutzerfreundlichkeit steigern, in dem wir die Eingabe der zu analysierenden Sätze und deren Notation als link/3-Fakten automatisieren.

Aufgabe 1.

Legt eine Datei namens psg1.pl an.
Lasst von dort aus lexikon.pl aufrufen.
Setzt die Grammatik der vorherigen Seite psg1.pl um

Aufgabe 2.

Ergänzt die Datei psg1.pl um Phrasenstrukturregeln, die auch die Analyse der folgenden Sätze ermöglichen und überprüft anschließend, ob die Sätze akzeptiert werden.:
Bart slept
The dog bit Lisa
Barney sold the beer to Homer
Marge read a rather silly cartoon.
Tip: Überlegt Euch zunächst, welche PS-Regeln für diese Sätze nötig sind, und übersetzt diese dann anschließend in Prolog. In den Fällen, in welchen neue Wortarten auftreten (Eigennamen, Präpositionen usw.), müssen natürlich auch entsprechende Regeln für die lexikalischen Kategorien formuliert werden! Wenn die Analyse nicht klappt, kann es auch immer am Lexikon liegen

Aufgabe 1.

Legt eine Datei namens psg1.pl an.
Lasst von dort aus lexikon.pl aufrufen.
Setzt die Grammatik der vorherigen Seite psg1.pl um

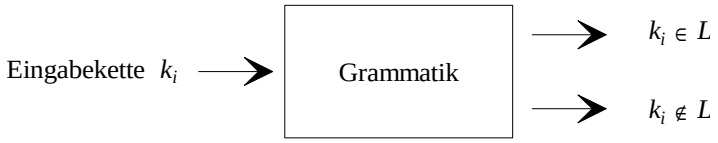
Aufgabe 2.

Ergänzt die Datei psg1.pl um Phrasenstrukturregeln, die auch die Analyse der folgenden Sätze ermöglichen und überprüft anschließend, ob die Sätze akzeptiert werden.:
Bart slept
The dog bit Lisa
Barney sold the beer to Homer
Marge read a rather silly cartoon.
Tip: Überlegt Euch zunächst, welche PS-Regeln für diese Sätze nötig sind, und übersetzt diese dann anschließend in Prolog. In den Fällen, in welchen neue Wortarten auftreten (Eigennamen, Präpositionen usw.), müssen natürlich auch entsprechende Regeln für die lexikalischen Kategorien formuliert werden! Wenn die Analyse nicht klappt, kann es auch immer am Lexikon liegen

Teil 2: Die Ausgabe der Strukturbeschreibung

Die PSG-Grammatik hat bis jetzt den Status eines Akzeptors – sie erkennt, ob eine gegebene Wortkette mit Bezug auf die Grammatik ein Satz (bzw. eine NP oder eine VP usw.) ist, oder nicht, und sie gibt dafür die Positionen an.

Etwas abstrakter betrachet: wenn wir eine Grammatik einer Sprache *L* als eine Ein-Ausgabe-Vorrichtung auffassen, die Ketten aus einem gegebenen Vokabular als Eingabe erhält und als Ausgabe die Entscheidung liefert, ob die Eingabe ein Satz von *L* ist oder nicht (Werte: {ja, nein}, oder {wahr, falsch}), haben wir es mit einer Erkennungsgrammatik zu tun.



Erkennungsgrammatik

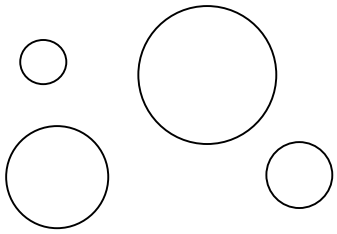
Interessanter als Erkennungsgrammatiken sind jedoch Grammatiken, die darüber hinaus jedem erkannten Satz eine bzw., bei Ambiguität, mehrere Strukturbeschreibungen zuordnet:

Generative Grammatik

Eine Grammatik, die über die bloße Entscheidung über die Wohlgeformtheit hinaus jedem erkannten Satz eine Menge von Strukturbeschreibungen zuordnet, ist eine generative Grammatik. An dieser Stelle kommt einem vielleicht der folgende Gedanke: Moment mal – generative Grammatik – das ist doch Chomsky-Grammatik? Generative Transformationsgrammatik, Government & Binding, Minimalist Program und so?!? Es ist in der Tat so, dass die Bezeichnung 'generative Grammatik' oft synonym mit einer der Modellvarianten bzw. dem gesamten Ansatz aus dem chomskyschen Dunstkreis verwendet wird. Das Attribut 'generativ' soll hier aber einerseits allgemeiner, andererseits auch eingeschränkter verstanden werden, als es häufig der Fall ist. Allgemeiner deshalb, weil es auf eine ganze Reihe von Grammatiktypen anwendbar ist, denen allesamt gemein ist, dass sie Verfahren für die Erzeugung der

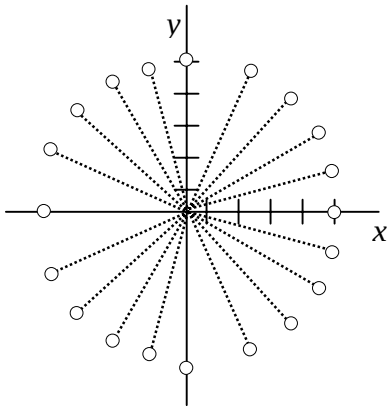
Strukturbeschreibung von Sätzen zur Verfügung stellen. Dazu gehören – neben den oa. Grammatikmodellen – auch Ansätze aus dem Bereich der Lexikalisch-Funktionalen-Grammatik (LFG); der Headdriven-Phrase-Structure-Grammar (HPSG); der Kategorialgrammatik usw. 'Generativ' ist andererseits eingeschränkt, weil es hier nur **mittelbar** um die wie auch immer geartete Produktion konkreter Sätze geht (und darum ging es auch Chomsky nie, was in der Vergangenheit aber häufig fehlinterpretiert wurde). Eine generative Grammatik kann durchaus eingesetzt werden, um über die Erzeugung von Strukturbeschreibungen konkrete Sätze zu generieren, was man mit Prolog auch einfach nachprüfen kann. Dieser Punkt ist aber im Grunde ein reines Nebenprodukt der Art und Weise, in der Aussagen über das Konstrukt 'Satz' formuliert sind. Während eine Erkennungsgrammatik einer eingegebenen Wortkette Grammatikalität in einer Sprache *L* attestiert (oder nicht), muss eine generative Grammatik qua ihres Aufbaus in der Lage sein, über das System der Regeln alle potentiellen Sätze der Sprache *L* zu generieren. Diese Menge potentieller Sätze ist natürlich, wie wir wissen, unendlich. Ergo kann es gar nicht darum gehen, tatsächlich alle Sätze zu generieren (im Sinne von 'produzieren'); sondern nur darum, ein System zu erstellen, über welches diese Menge in ihren Eigenschaften vollständig beschrieben ist. 'Generativ' darf hier also nicht im Sinne von 'produzieren', sondern sollte im Sinne von 'beschreiben' verstanden werden.

Nehmen wir dazu ein Beispiel aus einem ganz anderen Bereich. Die folgenden Objekte gehören allesamt zur gleichen Klasse, nämlich zu den Kreisen:



Kreise

Um diese Objekte einigermaßen explizit zu beschreiben und sie von Objekten, die keine Kreise sind (wie z.B. Rechtecken) abzugrenzen, reichen Aussagen wie 'Kreise sind rund' oder gar 'Kreise sind kreisförmig' natürlich nicht aus. Eine Möglichkeit, das Konstrukt 'Kreis' präzise zu beschreiben, besteht darin, die in der Geometrie verwendete Definition für Kreise heranzuziehen. Demnach ist der Kreis der geometrische Ort aller Punkte, für die gilt, dass sie in derselben Entfernung (= Radius) zu einem gegebenen Punkt (= Mittelpunkt) in einer Ebene liegen. Mit Bezug auf ein zweiachsiges Koordinatensystem und Kreise, die um den absoluten Nullpunkt liegen, kann diese Aussage wie folgt formalisiert werden: $x^2 + y^2 = r^2$. Der Radius *r* des Kreises entspricht mithin der Wurzel aus der Summe von x^2 und y^2: $r = \sqrt{x^2 + y^2}$. Wenn wir diese Formel anhand spezifischer Zahlenwerte konkretisieren, beispielsweise $x = 3$ und $y = 4$, liefert sie uns sie uns für *r* den Wert 5 (die Wurzel aus der Summe von 9 und 16, also 25, ist 5). D.h. dass die Gesamtheit aller Punkte, die im Radius 5 um den Nullpunkt liegen, ein geometrisches Objekt vom Typ 'Kreis' beschreiben. Exemplarisch soll das in der nachfolgenden Graphik dargestellt werden: [30]



[30] Das Koordinatensystem ist handgemalt und deshalb unpräzise. Für die Mathe-Nachhilfe vielen Dank an Prof. Wagner

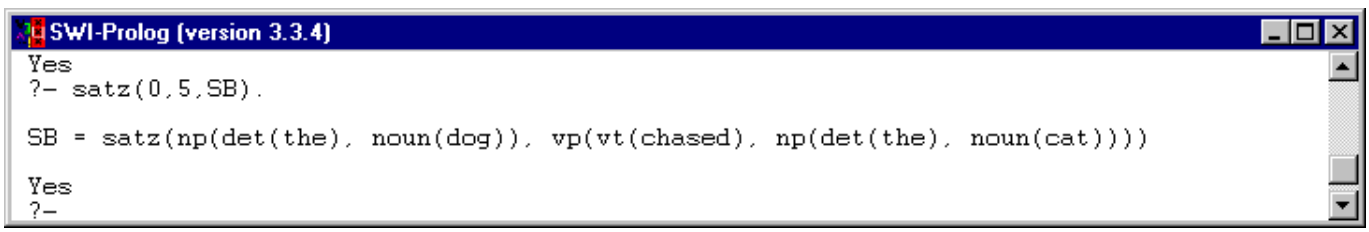
Über diese Definition ist das Konstrukt 'Kreis' (mit Bezug auf andere, unbhängig definierte theoretische Konstrukte wie 'Punkt' einerseits und bestimmte Operationen wie z.B. 'Addition' andererseits) präzise definiert. Mit entsprechendem Werkzeug, z.B. Lineal und Zirkel, könnte man die Definition dazu benutzen, alle potentiell möglichen Kreise zu erzeugen: dadurch, dass man jeweils unterschiedliche Werte für die Variablen *x* und *y* einsetzt, lassen sich Kreise in beliebiger Größe generieren. In der Tat kann eine Formel wie die oben angegebene – beispielsweise in *C*(omputer)-*A*(ssisted)-*D*(esign)-Programmen – genau für diesen Zweck eingesetzt werden. Sie ist aber dennoch keine 'Kreisgenerierungsformel' in dem Sinn, dass es ihre exklusive Aufgabe wäre, die Menge aller Kreise zu produzieren, denn auch hier gilt - genau wie bei den Sätzen einer Sprache L – dass diese Menge undendlich ist. Die oa. Defintion liefert stattdessen eine explizite Beschreibung des (geometrischen) Konstruktes 'Kreis'.

Die Analogie zu den Ausführungen über generative Grammatik sollte klar sein – hier geht und ging es auch früher nicht darum, eine Grammatik im Sinne einer 'Satzerzeugungsmaschine' zu erstellen, die mehr oder weniger undifferenziert Wortketten ausspuckt, die in einer Sprache L den Status von Sätzen haben, sondern stattdessen um eine explizite Beschreibung des Konstruktes 'Satz'. Diese explizite Beschreibung erfolgt in einer PSG als generativer Grammatik mit Bezug auf andere, unabhängig definierte theoretische Konstrukte wie z.B. 'NP' oder 'Präposition' einerseits und bestimmte Operationen, wie z.B. Phrasenstrukturregeln andererseits.

Diese ganze Herangehensweise an den Umgang mit Sprache ist natürlich entscheidend von dem Bestreben beeinflußt, diejenigen Aussagen, die man über den Gegenstandsbereich (im vorliegenden Fall eben die Syntax einer Sprache *L*) macht, so zu halten, dass sie den Kriterien für Wissenschaftlichkeit und Präzision genügen, wie z.B. intersubjektive Überprüfbarkeit, Eindeutigkeit, Widerspruchsfreiheit, Transparenz der Argumentation usw. Dieses Bestreben verdient natürlich, gewürdigt zu werden, die konkrete Umsetzung kann aber durchaus auch hinterfragt werden. Um das Konstrukt 'Satz' in einer Sprache *L* nach dem Muster wie in der hier verwendeten PSG zu definieren, werden etliche Abstraktionen getroffen. Man bezieht sich in der Grammatik auf lexikalische und syntaktische Kategorien und ggf. auf die Flexionsmorphologie, aber bestimmte andere Aspekte werden dabei völlig ausgeblendet – z.B. Fragen der Semantik oder Pragmatik. Satzübergreifende syntaktische Phänomene können mit der von uns verwendeten PSG ebenfalls nicht erfasst werden. Genau diese Arten von Information sind in anderen Ansätzen (und auch in moderneren Formen der generativen Grammatik chomsyischer Prägung) von vorneherein viel systematischer in die Grammatik integriert. Soviel erstmal zu einer Einschränkung des Begriffes 'generativ Grammatik' – damit könnte man allerdings Jahrzehnte zubringen.

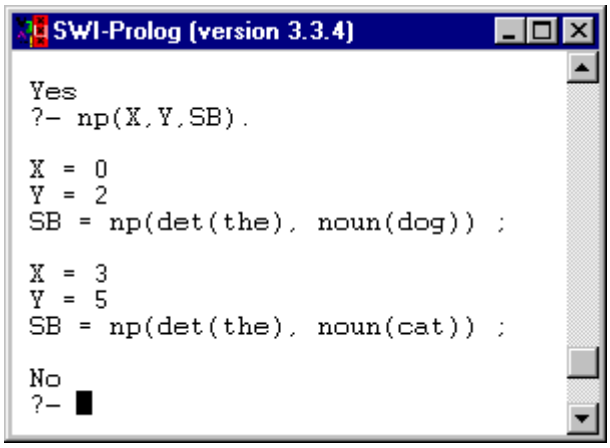
Was die Implementierung einer generativen Grammatik in Prolog betrifft (hier auf Grundlage einer PSG, also eines Ansatzes, der in der Tat mit dem Hause Chomsky assoziiert ist), zuerst einmal eine gute Nachricht: im Kern, d.h. vom gesamten Programmaufbau her, haben wir die dafür notwendigen Elemente bereits in Teil 1 dieses Kapitels, also in dem PSG-Akzeptor angelegt. In der Tat wird eine Wortkette ja mit Bezug auf Phrasenstrukturregeln – und nicht etwa auf so etwas wie die Satzmuster im ersten Teil des 6. Kapitels – analysiert. Das Ergebnis dieser Analyse, sprich die Aussagen 'X ist ein Satz / X ist kein Satz' rekurriert auf die Grammatik, das heißt der Satz entspricht einer der in der Grammatik explizit beschriebenen syntaktischen Strukturen. Diese aber werden bei der Analyse nicht angezeigt. Um dieses zu bewerkstelligen, werden alle Regeln für die syntaktischen und die lexikalischen Kategorien renotiert. Diese Modifikation läuft daraufhinaus, dass wir diese Regeln, genauer gesagt die Prädikate, um die es geht, also satz/2, np/2, vt/2, det/2 usw. usf., um ein drittes Argument erweitern, und genau dieses Argument wird auf die Strukturbeschreibung der Wortketten in question abgebildet.

Eine Eingabe wie satz(0,5,SB), wobei das dritte Argument, die Variable 'SB', für 'Strukturbeschreibung' steht, soll die folgende Ausgabe erhalten:



Auf die Tatsache, dass eine Funktor-Argumentstruktur bzw. ein Klammerausdruck wie in dem obigen Screen-Shot 1-zu-1 auf einen Baumgraphen abgebildet werden kann, wird hier mit keinem Wort eingegangen, denn das ist Syntax-Basiswissen.

Auch andere Anfragen sind nun möglich, beispielsweise nach den Strukturbeschreibungen der Nominalphrasen, hier mit Variablen für die Positionen:



Wie setzt man das konkret um? Am Beispiel der Regel für den Satz soll dieser Vorgang erläutert werden. Die bisherige Regel für einen Satz lautete:

```
satz(P0,Pn):-
        np(P0,P1),
        vp(P1,Pn).
```

Der Regelkopf satz(P0,Pn) soll nun um ein Argument dahingehend erweitert werden, daß auch die Strukturbeschreibung des Satzes enthalten ist. Das Problem dabei ist: wir wissen ja beim Aufruf des Prädikates überhaupt nicht, wie diese Strukturbeschreibung auszusehen hat – diese muss bei der Abarbeitung der eingegebenen Wortkette ja erst generiert werden. Welche Form muss also das dritte Argument von satz/3 haben?

Nun, alles, was wir hier sagen können, ist die Tatsache, dass es sich bei der Strukturbeschreibung des Satzes um eine Funktor-Argument-Struktur handeln wird, deren Funktor 'satz' ist und deren Argumente

a) die Strukturbeschreibung der (Subjekts-) NP, die erst ermittelt werden muss und

b) die Strukturbeschreibung der VP, die ebenfalls erst ermittelt werden muss, sind

Wie diese Strukturbeschreibungen intern genau aussehen ist nicht bekannt, sondern muss erst generiert werden. Die folgenden Beispiele zeigen, dass die NP und VP jeweils ganz unterschiedlich aufgebaut sein können:

{ *John* / *The boy* / *A student with glasses* } { *slept* / *read a book* / *put the money on the counter* }

Um diesen Sachverhalt in Prolog umzusetzen, wird das dritte Argument von satz/3 in einer Funktor-Argumentstruktur notiert, in der der Funktor 'satz' lautet (soviel ist ja klar) und dessen Argumente als Variablen auf die jeweiligen Strukturbeschreibungen von Subjekts-NP und VP verweisen.

```
satz(P0,Pn, satz(NP,VP)):- np(P0,P1,NP), vp(P1,Pn,VP).
```

Strukturbeschreibung des Satzes in Form einer Struktur mit dem Funktor *satz* und den Strukturbeschreibungen der NP und der VP als Argumenten.

Struktur-beschreibung der NP

Struktur-beschreibung der VP

Natürlich müssen wir die Regeln für die NP und die VP ebenfalls um ein drittes Argument anreichern. Im Einzelnen sieht das so aus:

- die Strukturbeschreibung der Kategorie NP besteht aus den jeweiligen Strukturbeschreibungen des Determinators und des Nomens:

```
np(P0,Pn,np(Det,Noun)):-
        det(P0,P1,Det),
        n(P1,Pn,Noun).
```

- die Strukturbeschreibung der Kategorie VP besteht aus den jeweiligen Strukturbeschreibungen des Verbes und der NP:

```
vp(P0,Pn,vp(VT,NP)):-
        vt(P0,P1,VT),
        np(P1,Pn,NP).
```

Soviel zu den Regeln für die syntaktischen Kategorien.

Die Regeln für die lexikalischen Kategorien det, noun und vt sahen ja etwas anders aus als die für die phrasalen Kategorien:

```
det(P0,P1):- link(P0,Wort,P1), lex(Wort,det).
noun(P0,P1):- link(P0,Wort,P1), lex(Wort,noun).
vt(P0,P1):- link(P0,Wort,P1), lex(Wort,vt).
```

Auch hier müssen die Regelköpfe um ein Argument erweitert werden. Der Unterschied zu den phrasalen Kategorien und dem Satz besteht aber darin, daß sich die Strukturbeschreibung hier nicht aus zwei anderen Strukturbeschreibungen zusammensetzt – mit den lexikalischen Kategorien ist sozusagen das Ende der Verzweigungen erreicht.

Anders ausgedrückt: die Strukturbeschreibung einer lexikalischen Kategorie besteht nur in der Angabe der lexikalischen Kategorie und dem Lexem, durch welches diese repräsentiert ist. Die Regeln für die lexikalischen Kategorien müssen also wie folgt erweitert werden:

```
det(P0,P1,det(Wort)):-
        link(P0,Wort,P1),
        lex(Wort,det).
```

Hier gibt der Funktor Aufschluß über die lexikalische Kategorie, das Argument ist das Wort selbst.

```
noun(P0,P1,noun(Wort)):-
        link(P0,Wort,P1),
        lex(Wort,noun).
vt(P0,P1,vt(Wort)):-
        link(P0,Wort,P1),
        lex(Wort,vt).
```

Wenn die Grammatik auf diese Art geändert wird, gibt Prolog für die jeweiligen syntaktischen Kategorien eine Strukturbeschreibung in Form eines Klammerausdruckes aus. Diese Strukturbeschreibungen werden bei Abarbeitung der entsprechenden Anfragen generiert, und es sollte mithin klar werden, wie genau das Verhältnis zwischen der Grammatik auf der einen und den Strukturbeschreibungen auf der anderen Seite ist – hier sieht man ganz genau, wie diese aus der Grammatik abgeleitet werden. Diese schöne Transparenz ist bei manchen der modernen Grammatikmodellen nicht mehr in dieser Form gegeben, insofern sich diese von klar regelbasierten Systemen weg- und zu eher restriktionsbasierten Systemen hinbewegt haben (zumindest auf dem Papier, wenn man versucht, sie in einem Programm konkret umzusetzen, kann sich das das Bild wieder ändern). 'Restriktionsbasiert' (die englische Bezeichnung heißt 'constraint based') kann hier so verstanden werden, dass die Grammatik eine Reihe von Strukturprinzipien kodiert, die von einer Wortkette erfüllt sein müssen, damit diese als wohlgeformt 'durchgeht'. Diese Strukturprinzipien (die beispielsweise auch Bezug nehmen auf bestimmte semantische Aspekte) sind aber ganz anderer Natur und sehr viel abstrakter als beispielsweise PS-Regeln. Mit Bezug auf die Rektions- und Bindungstheorie kann dieser Aspekt wie folgt beschreiben werden:

> Thus rather than building conditions into the rules themselves (as in the standard theory), or even imposing general conditions on the operation and interaction of simplified rules within the system (as in the early stages of the extended standard theory), Chomsky has now developed a theory of conditions on *the*

> *representations themselves*. The rules [...] do no more than specify those aspects of representations that do not follow from general principle. (GEOFFREY HORROCKS, *Generative Grammar*, Longman 1987:98)

Diese interessante Entwicklung im Bereich moderner Grammatikmodelle kann an dieser Stelle weder genauer erklärt noch weiter nachgezeichnet werden. Interessierte Studierende werden an entsprechende Lehrveranstaltungen verwiesen.

Nun ist die Rohform unseres Programms erstellt – es akzeptiert eine grammatische Wortfolge als Satz und gibt die Strukturbeschreibung an. So richtig prall ist das Programm aber nicht – weder die umständliche Eingabe von Sätzen als link/3 Fakten noch die Form, in der die Strukturbeschreibungen angezeigt werden (als Klammerausdruck) erfreuen die Benutzer. Um die Revision dieser Mängel geht es in den nächsten Abschnitten.

Aufgabe 3

Lasst die Sätze
Lisa slept
John cried
The girl loved the dog
Some boy kicked the table
John tied the man with the rope
Mary gave the dog to the girls
Bart put the book under the table
A terribly ugly boy kicked the dog

derart analysieren, dass jeweils eine Strukturbeschreibung erzeugt wird. Bedenkt dabei, dass Ihr vor der Eingabe des jeweiligen Prädikats link/3 das "alte" Prädikat link/3 mit abolish/1 aus der Wissensbasis entfernen müsst. Bitte kein Gemaunze wg. der umständlichen Eingabe – die Änderung der Bequemlichkeit steht als nächtest auf der Agenda.