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.

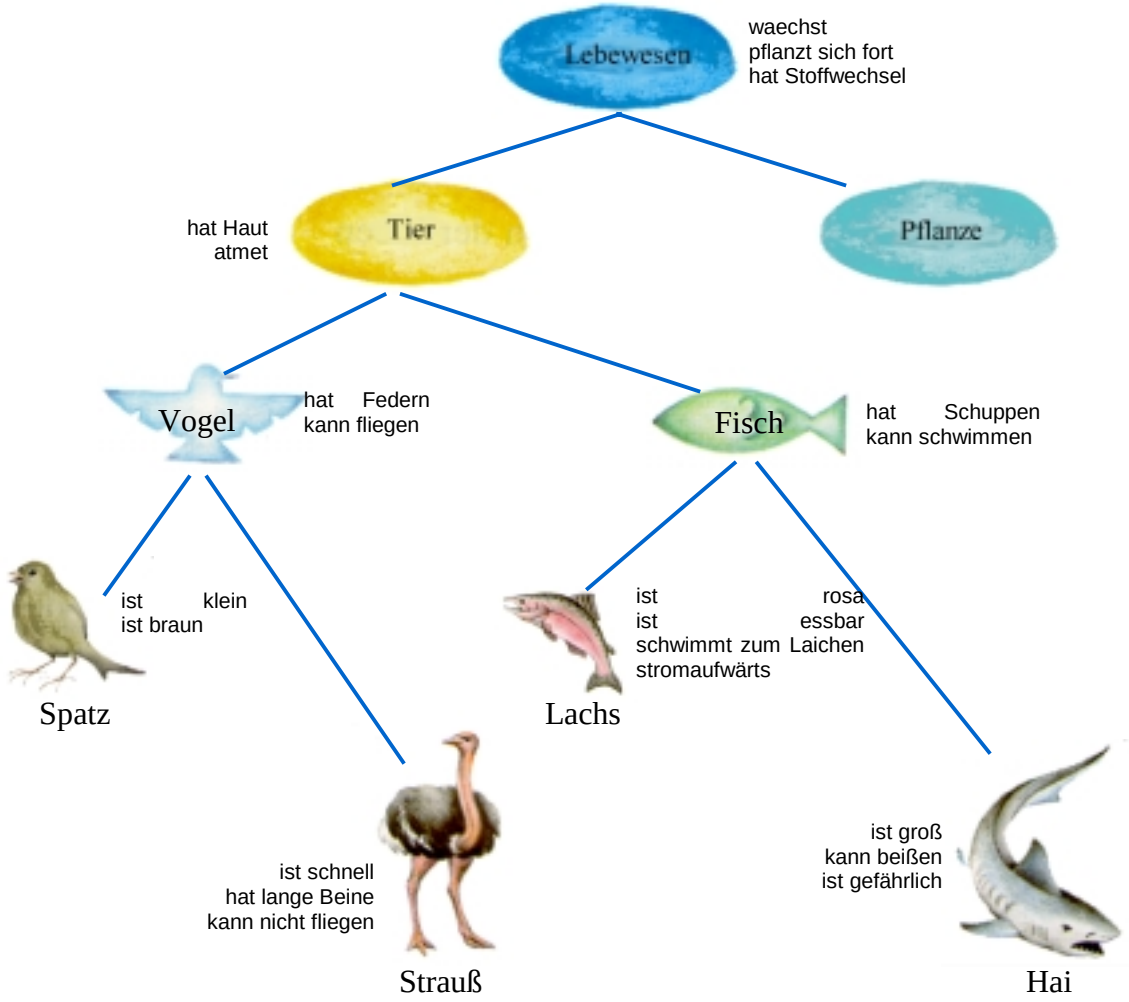


Abb. 3.1. Ausschnitt aus einem semantischen Netz.
(Abbildung nach George Miller, *Wörter. Streifzüge durch die Psycholinguistik*, Spektrum Verlag 1993)

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'.

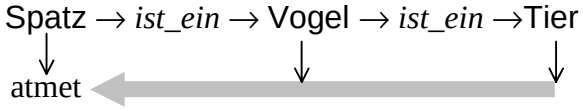
Im Rahmen der KI-Forschung sind semantische Netze spätestens seit den Arbeiten von M.R. Quillian Ende der sechziger Jahre ein Begriff.[11] Quillians zentrales Anliegen war es, ein Modell des menschlichen Gedächtnisses zu entwickeln. In Zusammenarbeit mit A. M Collins wurde eine Version dieses Modells 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).
```

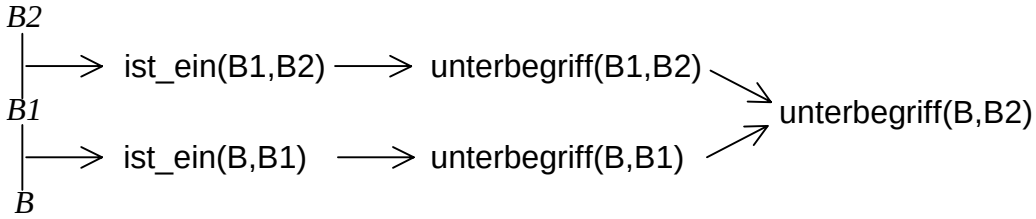
Damit erhalten wir aber eben nur die unmittelbaren Unterbegriffe. Betrachten wir nun eine Möglichkeit, sozusagen eine Stufe höher zu gehen:

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:

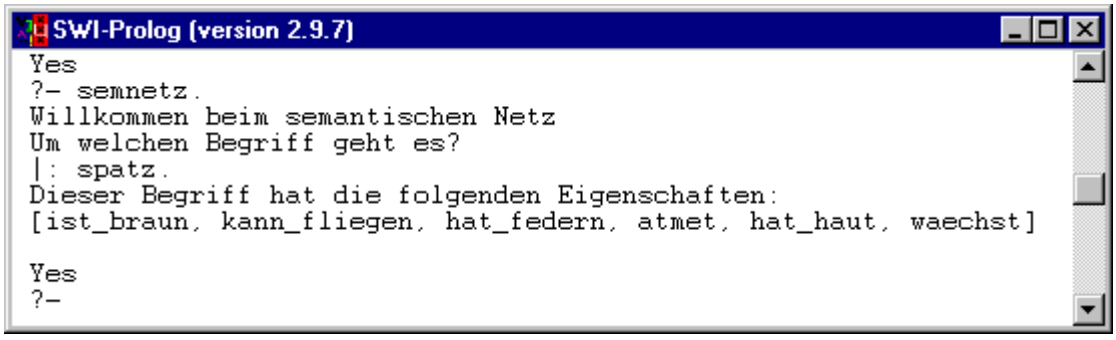
'Willkommen im semantischen Netz. Um welchen Begriff geht es?'

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:

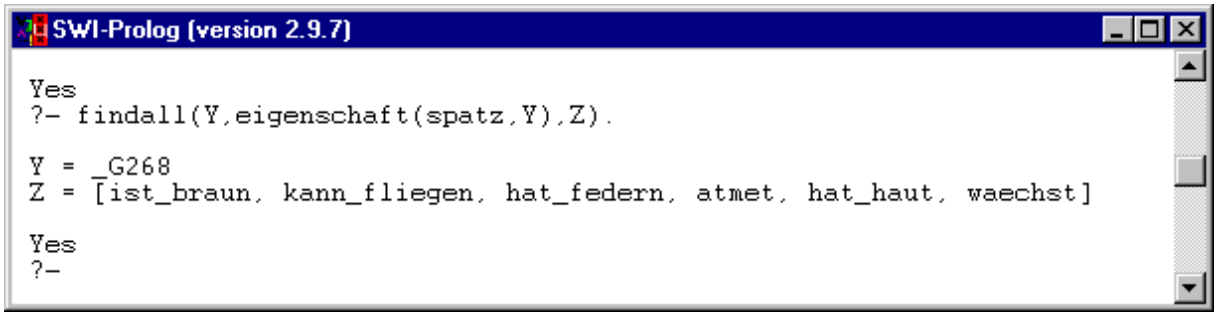


Was wir für diese Prozedur brauchen, sind die vordefinierten Prädikate write/1, read/1, nl, und findall/3.

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).
```

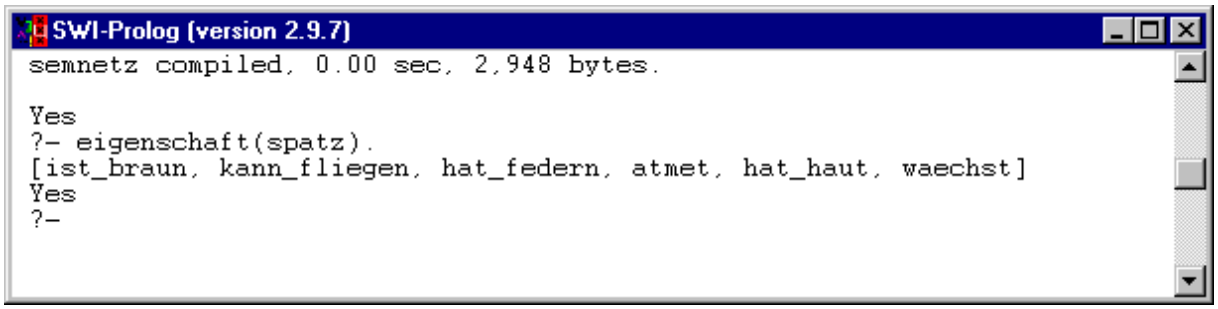
Hier gibt es aber ein Problem: Prolog wird durch nichts veranlaßt, dem Benutzer die Liste L der Eigenschaften auch tatsächlich mitzuteilen. Wird dieses Prädikat nicht umformuliert, so kommen als

[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).
```

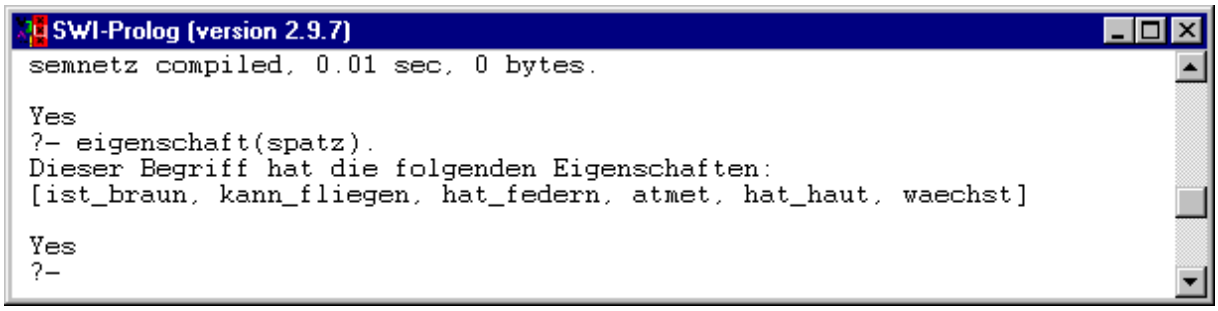
Das Resultat sieht bei entsprechender Anfrage (hier: eigenschaft(spatz)) doch schon viel ansprechender aus als die umständliche alte Version:



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.