2 Prolog

In dieser Vorlesung verwenden wir die Programmiersprache Prolog für alle Übungsaufgaben. Dafür gibt es mehrere Gründe. Erstens sind Prolog-Programme kompakt, so daß sie den Blick auf das Wesentliche nicht verstellen. Zweitens sind Prolog-Programme kurz genug, um oft auf eine Folie bzw. Tafelseite zu passen. Drittens kann man sehr schnell anfangen, in Prolog zu programmieren. Zwar gibt es ausgesprochen schwierige Prolog-Programme und sehr raffinierte Programmiertricks in Prolog, aber diese müssen nicht verstanden werden, bevor man anfangen kann zu programmieren. Mit den rudimentären Grundkenntnissen, die in diesem Kapitel vermittelt werden, können Sie sofort loslegen! Viertens sind in Prolog bereits einige Dinge eingebaut, die wir für die Übungsaufgaben benötigen. So ist ein Parser (syntaktische Analyse im Kapitel 6) fester Bestandteil von Prolog. Wir können also Grammatiken schreiben und brauchen dann nur den Prolog-Parser aufzurufen, um sie zu testen. In einer anderen Programmiersprache müßte zunächst der Parser implementiert werden, bevor eine kleine Grammatik geschrieben werden kann.

2.1 Formen

Manchmal wird Prolog als logischer Wissensrepräsentationsformalismus betrachtet. Die zugrundeliegende Logik ist die der Hornklauseln mit Resolutionsbeweis. Man sagt dann, daß Prolog Wissen über einen Sachbereich in Form von Fakten und Regeln darstellt. Eine Aufgabe wird durch eine Anfrage ausgedrückt. Die Anfrage wird von Prolog durch seine eingebaute Beweisstrategie anhand des Wissens beantwortet. Zusätzlich zur Logik enthält Prolog Kontrollelemente der Programmierung. Ich betrachte Prolog als Programmiersprache und wir können in dieser Programmiersprache Wissensrepräsentationen verschiedener Art realisieren. Im folgenden stelle ich zunächst die syntaktischen Formen von Prolog vor. Dabei gebe ich die logische Notation und die Prolog-Schreibweise an.

Literal: Ein **Literal** ist ein positives oder negatives Prädikat mit bestimmter Stelligkeit (Anzahl von Argumenten).

Ein Literal ist zum Beispiel: `mutter(X, Y)` ein anderes $\neg$ `vater(X, Y)`

In Prolog sind zwei Prädikate mit unterschiedlicher Stelligkeit und demselben Namen unterschiedliche Prädikate. Zum Beispiel sind

`kind(Kind, Vater, Mutter)` und `kind(Kind, Elternteil)`

die beiden verschiedenen Prädikate

`kind/3` und `kind/2`.

Klausel: Eine **Klausel** besteht aus Literalen, die disjunktiv (ODER) verknüpft sind.

Klausel: $(L_{1,1} \vee ... \vee L_{1,n1})$

In Mengenschreibweise sieht dieselbe Klausel so aus:

$\{L_{1,1}, ..., L_{1,n1}\}$

Hornklausel: Eine **Hornklausel** ist eine Klausel mit höchstens einem positiven Literal. Zum Beispiel:

{oma(X,Y), ¬ mutter(X,Z), ¬ mutter(Z,Y)}

Als **Regel** geschrieben sieht dieselbe Klausel so aus:

```
oma(X,Y) :- mutter(X,Z), mutter(Z,Y).
```

In Prolog wird also das positive Literal vor :- geschrieben, die negativen dahinter.

Anfrage: Eine **Anfrage** ist eine Hornklausel ohne positives Literal. Zum Beispiel:

{¬ mutter(X,Z), ¬ mutter(Z,Y)} oder {¬oma(X,Y)}

In der üblichen Prolog-Schreibweise sieht das so aus:

```
:- mutter(X,Z), mutter(Z,Y).
```
oder
```
:- oma(X,Y).
```

Fakt: Ein **Fakt** ist eine Hornklausel ohne negatives Literal. Es besteht also nur aus einem positiven Literal. Zum Beispiel:

```
{oma(X,Y)}.
```

In Prolog-Notation ist dies

```
oma(X,Y).
```

Prolog-Programm: Ein **Prolog-Programm** entspricht einer Formel in konjunktiver Normalform (d.h. ge-UND-ete ODER-Ausdrücke):

Programm: $(L_{1,1} \vee \ldots \vee L_{1,n1}) \& \ldots \& (L_{k,1} \vee \ldots \vee L_{k,nk})$

z.B.: (oma(X,Y) v ¬ mutter(X,Z) v ¬ mutter(Z,Y)) &

(oma(X,Y) v ¬ mutter(X,Z) v ¬ vater(Z,Y))

In Prolog wird das logische UND nicht aufgeschrieben. Es ist implizit gegeben.

```
oma(X,Y) :- mutter(X,Z), mutter(Z,Y).

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

Terme: Argumente eines Literals heißen **Terme**. Variablen, Zahlen, Konstante und Funktionen sind Terme. Das sind alle Terme. Wir schreiben Variablen mit Großbuchstaben oder beginnen sie mit _. Sowohl Konstante und Funktionen als auch Prädikatsnamen beginnen mit kleinen Buchstaben.

Funktionen sind meist arithmetisch und werden in einer Prolog-Bibliothek für mathematische Funktionen definiert. Meist wird eine Argumentstelle für den Eingabewert und eine für das Ergebnis reserviert. In der Dokumentation wird das Eingabeargument durch +, das Ausgabeargument durch - notiert. Zum Beispiel bedeutet die Funktion

```
sin(X,Y), daß Y=sin(X)
```

und wird in der Dokumentation angeführt als

```
sin(+,-)
```

Wenn man arithmetische Funktionen verwenden will, muß man am Anfang des Programms angeben:

```
:-ensure_loaded(library(math)).
```

Listen sind intern auch Funktionen. Das Funktionssymbol ist der Punkt. Beispielsweise wird die Liste `a, b, c` intern so dargestellt:

```
.(a, .(b, .(c, [])))
```

Die Funktion . hat die beiden Argumente `Kopf` und `Rest`.

```
.(Kopf, Rest)
```

Dabei besteht der `Rest` wieder in der Punkt-Funktion mit `Kopf` und `Rest`. Übersichtlicher geschrieben werden Listen durch eckige Klammern. Wenn man die ganze Liste schreiben möchte, wird sie einfach in die eckigen Klammern eingefaßt:

```
[a, b, c]
```

Will man nur jeweils mit dem Kopf oder dem Rest etwas anfangen, schreibt man:

```
[Kopf | Rest]

[a | Rest]
```

Statt wie oben für jedes Kind anzugeben, wer die Mutter ist, können wir auch mit der Listenfunktion für eine Mutter alle ihre Kinder angeben:

```
kinder (uta, [maria,mario]).
```

2.2 Arbeiten mit Prolog

Prolog kann interpretiert und compiliert verwendet werden. Wenn es interpretiert verwendet werden soll, ruft man als erstes Prolog auf.

```
prolog
```

Das Prolog-System meldet sich mit

```
?-
```

Dann kann etwas eingetragen werden. Zum Beispiel können Fakten und Regeln eingetragen werden. Das Eintragen von Fakten und Regeln erfolgt mit dem Prolog-Prädikat `assert/1`. Wenn etwas vorn vor die anderen Einträge angefügt werden soll, nimmt man `asserta/1`, wenn es hinter die anderen Einträge kommen soll, nimmt man `assertz/1`.

Beispiel:

```
?- assertz(vater(ulf, maria))
?- assertz(mutter(uta, mario)).
?- assertz(mutter(uta, maria)).
?- assertz(mutter(maria,anna)).
?- assertz((oma(X,Y):- mutter(X,Z), mutter(Z,Y))).
?- assertz((oma(X,Y):- mutter(X,Z), vater(Z,Y))).
```

Dies Verfahren empfiehlt sich zum Ausprobieren. Ansonsten wird Prolog in der compilierten Version genutzt. Man öffnet mit einem Editor eine Datei (z. B. mit dem Befehl `emacs verwandschaft.pl`) und schreibt dort direkt die Fakten und Regeln hinein.

Beispiel:

```
%Familie von Maria mit vater/2, mutter/2
vater(ulf,maria).
mutter(uta,mario).
mutter(uta,maria).
mutter(maria, anna).
oma(X,Y) :- mutter(X,Z), mutter(Z,Y).
oma(X,Y) :- mutter(X,Z), vater(Z,Y).
```

Wenn die Datei unter einem Namen mit der Extension `.pl` gesichert ist, ruft man Prolog auf und lädt die Datei mit dem Prädikat `consult/1`.

```
?- consult(verwandschaft.pl).
```

Dabei löscht `consult/1` alle Fakten und Regeln aus der aktuellen Datensammlung, die dasselbe Prädikat haben wie eines in der konsultierten Datei. Dies kann zu zwei Arten von Problemen führen. Zum einen können Klauseln verschwinden, die man gern behalten hätte. Nehmen wir an, ein Student und eine Studentin haben sich die Arbeit geteilt. Er hat die Familie von Ulf beschrieben, sie hat die Familie von Uta beschrieben. Er hat die Klausel für die Großmutter väterlicherseits, sie die Klausel für die Großmutter mütterlicherseits geschrieben. Ihre Datei heißt `sie.pl` und seine heißt `er.pl`. Nun wollen sie ihre Beschreibungen zusammenwerfen und rufen `consult(sie.pl)` auf und dann `consult(er.pl)`. Leider sind nun die Fakten über die Familie von Uta und die Klausel über die Großmutter mütterlicherseits fort. Entweder die Studierenden teilen sich die Arbeit, indem sie unterschiedliche Prädikate eintragen oder sie bringen ihre Beschreibungen im Editor zusammen.

Ein zweites Problem tritt immer wieder auf. Alles klappt prima an einem Tag, aber am nächsten Tag oder bei jemand anderem klappt dasselbe nicht. Hierfür ist der Grund oft, daß dort, wo das Programm erfolgreich läuft, noch Einträge im Prolog-System vorhanden sind (aus einer interpretierten Fassung oder einem vorher geladenen Programm), die das konsultierte Programm ergänzen. Beispielsweise könnte eine Studentin eine Datei `vatermutter.pl` geschrieben haben, die nur die Fakten, nicht jedoch die Regeln enthält. Die Regeln hat sie vorher in der interpretierten Fassung eingetragen. Nun kann Prolog die Großmutter von Anna ermitteln. Die Studentin schickt ihre Datei einem Kommilitonen. Bei ihm kann nun Prolog nicht die Großmutter von Anna ermitteln.

Wenn wir `verwandschaft.pl` vollständig geladen haben, können wir Anfragen stellen:

```
:- oma(Oma,anna).
Oma = uta
```

```
:- mutter(uta, X).
X= mario ;
X= maria
```

Durch die Eingabe des Semikolons hinter `X=mario` wird Prolog veranlaßt, nach weiteren Lösungen (`X=maria`) zu suchen.

Schließlich sei noch auf die Online-Hilfe in Quintus-Prolog hingewiesen. Mit dem Prädikat `help` erhält man eine kurze Bedienungsanleitung und mit dem Prädikat `manual/1` können Hilfetexte ausgewählt werden.

2.3 Beweisen

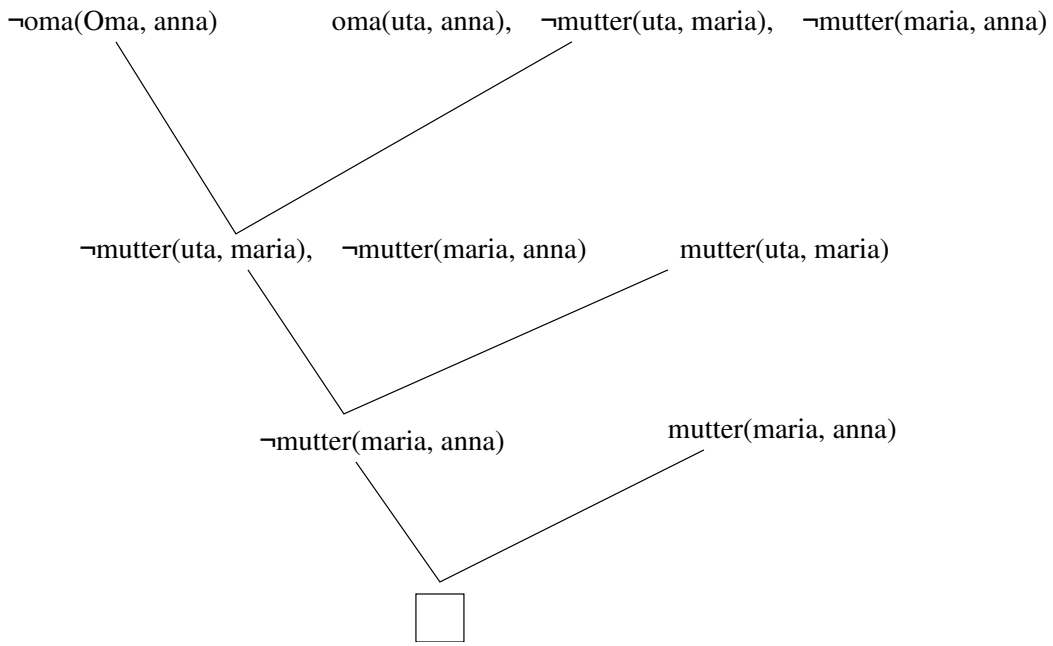
Prolog verwendet zur Beantwortung von Anfragen das Verfahren der **Resolution**.

Seien K_1 und K_2 Klauseln. Dann heißt R **Resolvent** von K_1 und K_2, falls es ein Literal L gibt mit $L \in K_1$ und $\neg L \in K_2$ und R die Form hat:

$$R = (K_1 - \{L\}) \cup (K_2 - \{\neg L\})$$

Dabei ist $\neg L$ definiert als $\neg A$, falls $L = A$, und als A, falls $L = \neg A$.

Die Resolution besteht also darin, ein positives und ein negatives Literal mit demselben Prädikat herauszuschneiden und die Reste der Klauseln zusammenzufügen (Schnittregel).



Eine Folge von Resolutionen, die zu der leeren Klausel führt, ist ein Beweis. Die leere Klausel (durch ein Quadrat dargestellt) ist immer falsch, so falsch wie ¬p & p. Mit den Resolutionen wird ein Widerspruchsbeweis geführt. Indem man das Gegenteil von dem, was man wissen möchte, widerlegt, beweist man das, was man wissen möchte. Deshalb sind Anfragen negative Literale. Es gibt bei demselben Programm und derselben Anfrage oft viele mögliche Folgen von Resolutionen. Es können auch mehrere Folgen von Resolutionen erfolgreich sein. Prolog hat eine Beweisstrategie, die die Reihenfolge der Beweisversuche festlegt. Es ist eine Rückwärtsverkettung von Beweiszielen.

Kontrollfluß:

Prolog arbeitet von links nach rechts.

1) Erst wird ein positives Literal gesucht, das zu der Anfrage paßt. Damit ist eine Klausel gefunden. Enthält diese Klausel nur das positive Literal (Fakt), dann ist der Beweis erfolgreich beendet. Sonst:

2) Es wird das am linkesten stehende negierte Literal dieser Klausel versucht, zu beweisen. Es wird also wie eine Anfrage behandelt (Schritt 1).

2a) Gelingt dieser Beweis, wird das direkt rechts danebenstehende negierte Literal der Klausel versucht, zu beweisen. Gibt es kein weiteres negiertes Literal in dieser Klausel, so ist der Beweis erfolgreich beendet.

2b) Gelingt der Beweis nicht, so wird zum nächstliegenden linken Literal zurückgegangen und ein anderer Beweis für dies Literal gesucht (Rückzug oder *backtracking*).

3) Wenn es keine Alternativen mehr gibt für ein Literal, das nicht bewiesen werden konnte, so gelangt man schließlich zum positiven Literal der Klausel zurück. Es wird dann ein anderes positives Literal gesucht, das zu der Anfrage paßt. Wenn es eines gibt, wird mit Schritt 2) fortgefahren. Wenn es kein anderes mehr gibt, ist der Beweis gescheitert. Prolog meldet dann `FAIL` und antwortet `no`.

Wenn wir den Befehl `trace.` eingeben, können wir den Kontrollfluß beobachten. Unser Beispiel ergibt bei der Anfrage `:-oma(Oma, anna).` etwa folgendes Bild.

Bei Schritt 1) meldet Prolog

```
CALL oma(_Oma,anna)
```

Die erste Klausel mit dem positiven Literal `oma` ist die Regel für die Großmutter mütterlicherseits. Jetzt wird - Schritt 2) - versucht, `mutter (_Oma, _V)` zu beweisen. Prolog meldet

```
CALL mutter (_Oma1, _V1).
```

Zuerst findet Prolog `mutter(uta,mario).` Prolog meldet

```
EXIT mutter(uta,mario).
```

Es wird dann das nächste Literal versucht zu beweisen. Prolog meldet

```
CALL mutter (mario, _W1).
```

Hier wird nichts passendes gefunden. Also wird noch einmal auf eine andere Art versucht, das nächst links stehende Literal zu beweisen. Prolog meldet

```
REDO mutter(_Oma2, _V2).
```

Es gibt einen alternativen passenden Fakt. Prolog meldet

```
EXIT mutter(uta,maria).
```

```
REDO mutter (maria, _W2).
```

```
EXIT mutter(maria,anna).
```

Da nun kein weiteres Literal mehr in der Klausel zu beweisen bleibt und alle Teilbeweise erfolgreich waren, meldet Prolog auch

```
EXIT oma(uta,anna)
```

Das Prolog-Protokoll (trace) ist hier etwas geschönt dargestellt. Schon an dem einfachen Beispiel sieht man, daß neue Variablennamen (mit _ vorweg) nötig waren, um nicht mit bereits in den Klauseln verwendeten Variablennamen durcheinander zu geraten. Für jeden Beweisversuch muß ein neuer Variablenname

eingeführt werden. Da dies sehr viele werden können und Prolog die Bedeutung der Klauseln nicht kennt, vergibt es intern Zahlen, die an das Zeichen _ angehängt werden.

Wie werden nun Literale, die Variable enthalten, mit anderen Literalen desselben Prädikat gleich gemacht? Man braucht einen Gleichmacher für Argumente eines Prädikats. In unserem Beispiel wurde `_W2` durch `anna` substituiert, so daß wir zwei bis auf ihr Vorzeichen gleiche Literale erhielten und schneiden (resolvieren) konnten.

Substitution: Eine **Substitution** ist eine endliche Menge $\{V_1/t_1, ..., V_n/t_n\}$, wobei $V_i \neq V_j$ für alle $i \neq j$. V_i/t_i bedeutet, daß die Variable V_i an den Term t_i gebunden wird. Eine Substitution anwenden, heißt, alle Vorkommen der Variablen innerhalb einer Klausel gleichzeitig durch den betreffenden Term zu ersetzen. Meist bezeichnet man eine Substitution mit σ. Die Identitätssubstitution ist die leere Menge.

`oma(Oma,anna)σ = oma(uta,anna)` mit `σ:{Oma/uta}`

Die Substitution macht ein Literal nie allgemeiner. Man darf also keinesfalls für eine Konstante oder Funktion eine Variable einsetzen! Nun wollten wir aber nicht irgendwelche Terme für irgendwelche Literale einsetzen, sondern mehrere Literale (bis auf ihr Vorzeichen) gleich machen. Eine gleichmachende Substitution heißt **Unifikation**.

Unifikator: Eine Substitution σ wird **Unifikator** genannt, wenn für eine endliche Menge von Literalen L die Anwendung der Substitution eine Menge mit nur einem Element ergibt. Die Literale aus L sind dann unifizierbar.

```
L: {vater(X,mario), vater(Y, mario), vater(ulf, Z)}

σ: {X/ulf, Y/ulf, Z/mario}

Lσ: {vater(ulf,mario)}   | Lσ | = 1
```

mgu: Ein Unifikator σ heißt **allgemeinster Unifikator** (mgu), wenn für jeden anderen Unifikator ρ eine Substitution τ existiert, so daß $\rho = \sigma\tau$. Mit anderen Worten: wenn man einen allgemeinsten Unifikator durch eine weitere Substitution spezialisiert, erreicht man einen anderen, nicht allgemeinsten Unifikator.

```
L:   {unterhalt(ulf, maria, X),
      unterhalt(Y,   maria, Z),
      unterhalt(Y,   maria, alimente(Y,V))}

σ: {Y/ulf, X/alimente(ulf,V), Z/alimente(ulf,V)}

ρ: {Y/ulf, X/alimente(ulf, 1000), Z/alimente(ulf, 1000)}

τ: {V/1000}
```

Der klassische Unifikationsalgorithmus findet für unifizierbare Literale immer den allgemeinsten Unifikator. (Der Beweis steht in Schöning 1995).

Unifikationsalgorithmus:

Eingabe: eine nicht-leere Menge L von Literalen.

$\sigma := \{\}$

solange $|L\sigma| > 1$:

Durchsuche die Literale in $L\sigma$ von links nach rechts, bis die erste Position gefunden wird, wo sich mindestens zwei Literale L_1 und L_2 unterscheiden.

Wenn keines der beiden sich unterscheidenden Zeichen eine Variable ist, halte an und gebe aus "nicht unifizierbar". Sonst:

Sei X die Variable und t der im anderen Literal beginnende Term:

Wenn X in t vorkommt, halte an und gebe aus "nicht unifizierbar". Sonst:

$\sigma := \sigma \{X/t\}$ ---- nacheinander σ und $\{X/t\}$ ausführen

σ als allgemeinsten Unifikator ausgeben und anhalten.

Da Namensgleichheit nur innerhalb einer Klausel auch Variablengleichheit bedeutet (während gleichbenannte Konstante und Funktionen im gesamten Programm gleich sind), werden in Prolog vor der Unifikation alle Variablen so umbenannt durch neue, noch nirgends vergebene Namen, daß gleiche Variablen innerhalb einer Klausel auch einen gleichen Namen erhalten, der sonst nicht auftritt. Nehmen wir unser Beispiel von den Unterhaltszahlungen und gehen davon aus, daß die Variablen bereits richtig behandelt wurden.

$|L\sigma| = 3$

1. Unterschied: `ulf, Y`

   $\sigma := \sigma$ {`Y/ulf`}

   $|L\sigma| = 3$

2. Unterschied: `X, Z, alimente(ulf,V)` -- muß aufgeteilt werden in:

   `X, alimente(ulf,V)` und `Z, alimente(ulf,V)`

   $\sigma := \sigma$ {`X/alimente(ulf/V), Z/alimente(ulf/V)`}

   $|L\sigma| = 1$

   σ: {`Y/ulf, X/alimente(ulf,V), Z/alimente(ulf,V)`}

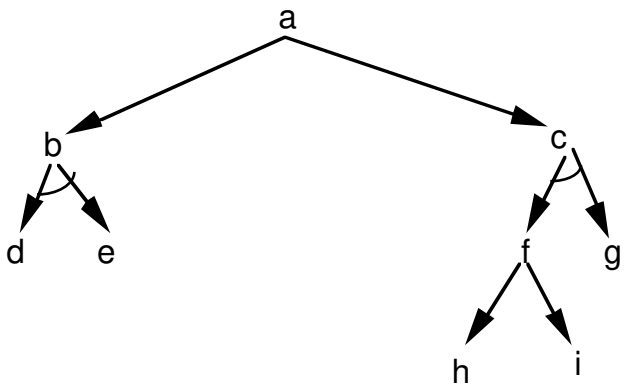
Dies Beispiel illustriert, was mit *beginnend* gemeint war: die ganze Funktion `alimente(ulf,V)` wird für die Variablen `X` und `Z` eingesetzt. Ein weiterer Unterschied besteht dann nicht mehr, so daß die Variable `V` erhalten bleibt und nicht etwa ein speziellerer Unifikator gewählt wird.

2.4 Prologs Kontrollstruktur als Tiefensuche in UND/ODER-Graphen

Wir können uns Prolog-Klauseln als ge-ODER-te Knoten in einem Graphen vorstellen, wobei die Literale in der Prämisse ge-UND-ete Unterknoten in dem Gra-

phen darstellen. Ein Beweis in Prolog kann dann als Tiefensuche in diesem Graphen dargestellt werden.[4]

Ein UND-ODER-Graph ist ein Graph, bei dem auf einer Ebene alle Nachfolgeknoten eines Knoten mit UND verknüpft und auf der nächsten Ebene alle Nachfolgeknoten eines Knoten mit ODER verknüpft sind. Der Weg vom Ausgangsknoten zum Ziel ist ein Baum, da die durch UND miteinander verknüpften Knoten alle zur Lösung gehören. Ein einfacher UND-ODER-Graph sieht z.B. so aus:



Dabei sind die Teilziele b und c alternativ, die Teilziele d und e müssen beide gezeigt werden, ebenso wie die Teilprobleme f und g beide gelöst werden müssen. Von den Teilzielen h und i muß nur eine gezeigt werden, egal welche. Wenn h und g Zielzustände ist, so gehören die Knoten a, c, f, g und h zum Lösungsbaum.

Tiefensuche durch einen UND-ODER-Graphen kann als Lösung eines Problems, das in Teilprobleme zerlegt wurde, ganz einfach folgendermaßen aufgefaßt werden (Bratko 1987:322):

Zur Lösung eines Problems - dargestellt durch einen Knoten K - benutze die folgenden Regeln:

1) Ist K ein Zielknoten, ist das Problem gelöst.

2) Hat K geODERte Teilprobleme als Nachfolger, dann löse eins von ihnen (versuche eins nach dem anderen, bis eins gelöst ist).

3) Hat K geUNDete Teilprobleme als Nachfolger, dann löse sie alle (versuche eins nach dem anderen, bis alle gelöst sind).

Während dieser Problemlösung muß der Lösungsbaum mitgeschrieben werden, damit man die Lösung auch verwenden kann. Wir nennen einen Knoten, dessen Nachfolgeknoten geODERt sind einen ODER-Knoten und den, dessen Nachfolgeknoten geUNDet sind, einen UND-Knoten.

Der Lösungsbaum enthält

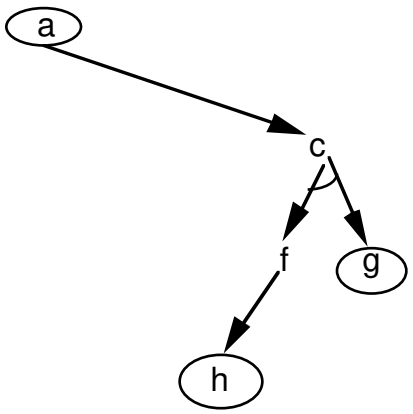
1) den Zielknoten,

2) für die Lösung verwendete ODER-Knoten mit dem jeweils an ihm hängenden Unterbaum,

[4] ACHTUNG: in der Klausel-Notation sind die Literale einer Klausel durch ODER verknüpft, die Klauseln eines Programms sind UND verknüpft. In der Graphen-Darstellung ist es genau anders herum!

3) für die Lösung verwendete UND-Knoten mit einer Liste von jeweils an ihm hängenden Unterbäumen.

Der Lösungsbaum des obigen Beispiels sieht so aus:



Tatsächlich gibt Prolog nicht den Beweisbaum aus, sondern nur die in ihm gefundenen Substitutionen für die Variablen im Zielausdruck.

2.5 Programmieren

Die Beweisstrategie von Prolog und die Variablenbindung durch Unifikation hat einige Effekte, die vielleicht nicht offensichtlich sind. Deshalb gehe ich in diesem Abschnitt kurz darauf ein. Zum anderen gibt es zwei Methoden, wie die Beweisstrategie von Prolog beeinflußt werden kann. Auch diese will ich vorstellen. Alle weiteren Programmiertricks lernt man am besten durch eigenes Ausprobieren und Erfahrung. Sie sind aber für die Übungsaufgaben nicht nötig.

Ein wichtiger Effekt der Prolog-Strategie ist, daß dasselbe Prädikat zum Testen und zum Generieren benutzt werden kann, je nachdem, wo die Variablen stehen. Nehmen wir für das Aneinanderhängen von Listen ein dreistelliges Prädikat an, dessen erste beiden Argumente die aneinanderzuhängenden Listen und das dritte Argument das Ergebnis darstellt.

```
append([1,2],[3,4],X)
```

liefert uns wie erwartet

```
X=[1,2,3,4]
```

Wir können aber auch eine Zerlegung suchen, indem wir

```
append(X, Y, [1,2,3,4])
```

aufrufen. Wir erhalten

```
X=[], Y=[1,2,3,4] ;
X=[1], Y=[2,3,4] ;
X=[1,2], Y=[3,4] ;
X=[1,2,3], Y=[4] ;
X=[1,2,3,4], Y=[]
```

Natürlich können wir auch

```
append(X,Y,Z)
```

aufrufen -- Prolog garantiert nicht, daß es anhält! Es ist die Aufgabe der Programmiererin oder des Programmierers, sich zu überlegen, was sie bzw. er

schreibt. Normalerweise hält Prolog aber nach der ersten gefundenen Lösung an: es wurde ein Beweis gefunden. Will man alle möglichen Beweise sehen, muß man einfach behaupten, der Beweis wäre nicht gelungen. Wenn ein Beweis nicht gelungen ist, sucht Prolog nach einer Alternative. Man teilt Prolog das Fehlschlagen eines Beweises durch `fail` mit. Wer also unbedingt Rechenzeit verbrauchen möchte, erreicht dies durch das folgende kleine Programm:

```
endlos :- append (X, Y, Z), fail.
```

Nun werden alle Listen erzeugt, weil nach jeder erfolgreichen Substitution der Variablen durch eine Liste der Beweis als gescheitert erklärt wird und Prolog nach einer Alternative sucht. Wenn der Wertebereich nicht unendlich ist wie die Menge aller Listen, so kann dies Vorgehen durchaus sinnvoll sein, um alle Lösungen zu finden. Wollen wir zum Beispiel alle Kinder von `maria` finden, so schreiben wir:

```
alle_kinder(X,Y):-mutter(X,Y),write('Kind='),write(Y),fail.
```

Dies würde auch alle Mütter liefern, falls es mehrere gäbe. Am Beispiel unserer kleinen Datei in Abschnitt 2.1 liefert der Aufruf

```
alle_kinder(uta,X).

Kind=mario Kind=maria

no
```

Manchmal will man den Kontrollfluß aber auch dahingehend verändern, daß keine weiteren Alternativen ausprobiert werden. Wenn man schon weiß, daß es maximal einen Beweis für ein Teilziel geben kann, setzt man hinter dieses Teilziel ein Ausrufezeichen (cut). Solange das Ausrufezeichen nicht von links nach rechts überschritten wurde, wird nach Alternativen gesucht. Danach werden mit der gefundenen Belegung die weiter rechts stehenden Teilziele versucht zu beweisen. Gelingt dies nicht, wird für den Beweis vor dem Ausrufezeichen keine Alternative gesucht, sondern abgebrochen. Dadurch werden Programme schneller. Diese Strategiemodifikation will sehr gut überlegt sein! Wenn wir beispielsweise schreiben

```
oma(X,Y):- mutter(X,Z),!,mutter(Z,Y).
```

weil wir uns dachten, daß eine Frau ohne Kind nicht Oma sein kann (richtig), so probieren wir in unserem Beispiel leider nicht mehr aus, ob Maria ein Kind hat. Nachdem die Belegung `mario` für `Z` gefunden wurde, beendet Prolog den Beweis erfolglos. Fälschlicherweise haben wir angenommen, daß das zuerst gefundene Kind auch Mutter ist, wenn überhaupt ein Kind Mutter ist. Oft ist man sich solcher impliziten Annahmen aber nicht bewußt. Also, Vorsicht mit dem Ausrufezeichen!

Abschließend noch ein Absatz zur Rekursion. Dies ist das wichtigste Konzept bei der Programmierung in Prolog. Man muß dabei aber beachten, daß Prolog von links nach rechts arbeitet. Der Rekursionsaufruf sollte also hinten in der Klausel sein.

```
liste_bearbeiten([Kopf|Rest], [Ergebnis|Ergebnis_rest]):-
      bearbeiten(Kopf,Ergebnis),
      liste_bearbeiten(Rest, Ergebnis_rest).
```

Wir haben jetzt sichergestellt, daß bei jedem Durchlauf der Klausel die Liste des Aufrufs verkürzt wird. Gleichzeitig bauen wir die Ergebnisliste auf. Leider haben wir davon noch nichts, wenn wir nicht eine Abbruchsbedingung schreiben. Um Rechenzeit zu sparen, berücksichtigen wir, daß Prolog die Klauseln von oben nach unten durchgeht. Wir schreiben also immer als erste Klausel für ein Prädikat den Zustand auf, den wir erreichen wollen. Wann ist das Problem gelöst? Was wollen wir mit der Lösung tun?

```
liste_bearbeiten([], Ergebnis):- write(Ergebnis).
```

Hier wollen wir, daß die zu bearbeitende Liste leer ist. Dann ist alles bearbeitet und das Problem ist gelöst. Die Lösung wollen wir ausgegeben haben, deshalb lassen wir sie ausgeben. Wir schreiben diese Klausel vor die andere. Prolog schreibt das Ergebnis jetzt in eine Zeile. Wenn wir jedes Teilergebnis in einer eigenen Zeile haben wollen schreiben wir:

```
liste_bearbeiten([], _).      % Ergebnis interessiert uns hier
                              % nicht mehr.

liste_bearbeiten([Kopf|Rest], [Ergebnis|Ergebnis_rest]) :-
      bearbeiten(Kopf,Ergebnis),
      write(Ergebnis), nl,    % jedes Teilergebnis wird
                              % ausgegeben und danach eine neue
                              % Zeile angefangen.
      liste_bearbeiten(Rest, Ergebnis_rest).
```

2.6 Literatur

Bratko, Ivan (1988): PROLOG Programmierung für künstliche Intelligenz, Bonn [u.a.]: Addison Wesley

Schöning, Uwe (1995): Logik für Informatiker, Heidelberg: Spektrum Akad. Verl.

Robinson, J.A. (1965): A Machine Oriented Logic Based on the Resolution Principle, in: *Journal of the ACM*, 12, 1.

3 Wissensrepräsentation

Die Wissensrepräsentation ist sicherlich das Kernstück der KI. Wir wollen Wissen über einen Sachbereich formalisieren und die Formalisierung zur Lösung von Problemen nutzen. Wenn man Prolog als Wissensrepräsentationsformalismus auffaßt (was wir nicht tun), so haben wir bereits einen solchen Formalismus und seinen Einsatz zur Problemlösung kennengelernt: Resolutionsbeweis bzw. Suche in UND-ODER-Graphen. Wir haben dabei gesehen, daß sowohl das Paradigma der Suche als auch das des Beweisens gute Möglichkeiten darstellen, Problemlösung zu beschreiben. In diesem Kapitel werden nun Repräsentationen und ihre Nutzung zum Problemlösen beschrieben. Prolog verwenden wir lediglich zur Programmierung dieser Formalismen und ihrer Interpreter.

Über Definitionen von Wissen und die Wissensebene wurde bereits im ersten Kapitel Newells Zuschreibung von Wissen aufgrund des Rationalitätsprinzips diskutiert. Hier soll nun noch ein weiteres Zitat angeführt werden, das deutlich macht, warum ich nicht nur "Hornklauseln" oder "logische Formeln" als Wissensrepräsentation angeführt habe, sondern "Hornklauseln mit Resolutionsbeweis":

> *"We can say metaphorically, that a book is a source of knowledge, but without the reader, the book is just ink on paper. Similarly, we often talk about the list and pointer data structures in an AI database as knowledge per se when we really mean that they represent facts or rules when used by a certain program to behave in a knowledgeable way." (Barr, Davidson 79)*

Wie immer Wissen im einzelnen definiert ist, **Wissensrepräsentation** ist dessen operationale Darstellung. Dabei wird eine Wissensrepäsentationssprache verwendet, um das Wissen über einen Sachbereich auszudrücken. In diesem Skript wie auch in der Literatur zum maschinellen Lernen ist die Wissensrepräsentationssprache verschieden vom Wissensrepräsentations formalismus. In anderen Artikeln wird oft Repräsentationssprache und Repräsentationsformalismus synonym gebraucht.

Die **Wissensrepräsentationssprache** entspricht der Signatur in der logischen Terminologie. Sie enthält also bereits die konkreten Prädikate oder Knoten- und Kantennamen oder Operatornamen, die für die Darstellung des Sachbereichs bzw. Problems verwendet werden.

Der Wissensrepräsentationsformalismus entspricht dem Kalkül in der logischen Terminologie. Ein **Wissensrepräsentationsformalismus** legt also fest, was wohlgeformte Ausdrücke sind und wie sie verarbeitet werden.

Schließlich ist ein **Wissensrepräsentationssystem** ein Programmsystem, das die Repräsentation und Verarbeitung von Wissen unterstützt. Ein Wissensrepräsentationssystem ist sachbereichsunabhängig. Es unterstützt den Aufbau und die Wartung aller Wissensbasen für Sachbereiche, die sich in dem entsprechenden Formalismus ausdrücken lassen.

3.1 Vollständige Operatoren und Suchverfahren

Die meisten Probleme, die mit KI-Verfahren gelöst werden, lassen sich als Suche im Problemraum beschreiben. Ein Suchproblem ist durch einen Anfangszustand, einen Zielzustand und Übergänge zwischen Zuständen beschrieben. Dadurch ist der Problemraum gegeben. Der **Problemraum** soll alle möglichen Lösungen enthalten. Ist die Lösung gar nicht in dem Problemraum vorhanden, kann sie durch kein Suchverfahren gefunden werden. Der Problemraum soll nicht einfach eine

ungeordnete Ansammlung aller für eine Klasse von Problemen möglichen Lösungen sein. Als abschreckendes Beispiel zu einem chaotischen aber vollständigen Problemraum gibt es die Geschichte von hundert Affen, die jeder wahllos auf einer Schreibmaschine tippen. Wenn sie lange genug tippen und zufälligerweise die entsprechenden Tasten in der entsprechenden Reihenfolge drücken, könnten sie alle Bücher des britischen Museums produzieren. Ein Algorithmus, der blind alle Möglichkeiten produziert, heißt deshalb "british museum algorithm". Damit ein günstiges Suchverfahren angegeben werden kann, strukturiert man den Problemraum, so daß die Zustandsübergänge die Zustände geeignet anordnen. Die Zustandsübergänge werden als Operatoren realisiert, die angeben, wie zu einem Knoten Nachfolger produziert werden. Man sagt, die Operatoren expandieren den Knoten. Vor der Expansion heißt ein Knoten "offen", danach "geschlossen". Meist werden Abbruchbedingungen bereits in die Operatoren einbezogen, so daß nur Nachfolgezustände erzeugt werden, die "legal" sind. Daher werden die Operatoren auch "legal move generator" genannt. Wir stellen uns den Problemraum als gerichteten Graphen vor, bei dem die Knoten die Zustände und die Kanten die Zustandsübergänge darstellen, die so gerichtet sind, daß sie von Anfangszuständen zu Zielzuständen führen.

Das **Suchverfahren** soll mit möglichst **wenig Aufwand** die Lösung finden. Das heißt, es soll möglichst schnell auf die Lösung stoßen, sie als solche erkennen und dann enden. Es soll möglichst nicht den gesamten Problemraum durchsuchen und als letztes erst die Lösung finden. Es soll auch möglichst nicht dieselben Bereiche des Problemraums mehrmals durchsuchen. Der Lösungsweg selbst soll möglichst kurz, minimal, sein. Das heißt, das Suchverfahren soll von einem Startknoten den **minimalen Pfad** zum Zielknoten finden. Zusätzlich können mit einer Kante von einem Knoten zu einem anderen Kosten verbunden sein. Das Suchverfahren soll dann nicht nur den kürzesten Pfad finden, sondern obendrein den **billigsten Pfad**. Wenn die Kanten keine unterschiedlichen Kosten haben, ist der kürzeste auch der billigste Pfad. Wenn wir also ein Problem als Suche formalisieren wollen, müssen wir uns um die folgenden Punkte kümmern:

- Problemraum:
  Was sind die Knoten, was sind die Kanten? Gibt es Zyklen in dem Graphen? Oder ist es ein Baum?

  Wie ist der Problemraum strukturiert, gibt es eine (partielle) Ordnung?

- Suchverfahren:
  Sollen zunächst alle Lösungsansätze gefunden werden (Breitensuche) oder erst ein Lösungsansatz zuende durchgegangen werden (Tiefensuche) oder soll jeweils vom meistversprechenden Knoten aus weitergesucht werden (heuristische Suche)?

  Wie kann man abschätzen, wie vielversprechend ein Knoten für das Weitersuchen ist?

- Abbruch der Suche:

  Unter welcher Bedingung ist eine Lösung gefunden (Zielzustand)? Unter welcher Bedingung kann die Suche abgebrochen werden, weil es (von diesem Knoten aus) keine Lösung mehr geben kann?

Knoten im Problemraum können mögliche Lösungen sein oder Zwischenergebnisse oder Zustände oder Problembeschreibungen. Entsprechend sind die Kanten Operatoren, die neue Knoten erzeugen, zusätzliche Hypothesen, elementare Handlungen oder Zusammenhänge. Diese verschiedenen inhaltlichen Interpretationen ändern nichts an den Suchverfahren. Sie bedeuten aber ein unterschiedliches Interesse: während man bei Knoten, die mögliche Lösungen reprä-

sentieren, nur an dem einen Ergebnisknoten interessiert ist, ist bei Zwischenergebnissen und Zuständen das Interesse auf den Pfad gerichtet. Bei Graphen, die Zusammenhänge widergeben, z.B. Probleme in Teilprobleme zerlegen, ist ein Teilgraph interessant.

Das bekannte Problem der Missionare und Kannibalen soll als Suchproblem formuliert werden:

> Drei Missionare und drei Kannibalen befinden sich an einem Flußufer. Alle wollen auf die andere Seite des Flusses. Sie haben ein Boot, das ein oder zwei Personen befördern kann. Wenn an einem Ufer mehr Kannibalen als Missionare sind, werden die Missionare verspeist. Wie bekommt man alle Personen auf die andere Seite des Flusses?

Der Anfangszustand ist also, daß alle Personen auf einer Seite des Flusses sind. Der Zielzustand ist, daß alle Personen auf der anderen Seite des Flusses sind. Die Zustandsübergänge sind Fahrten von einer Seite zur anderen mit ein oder zwei Personen. Der Zielzustand wird direkt erkannt. Operatoren, die die Personen einer Überfahrt vom einen Ufer abziehen und sie zu den Personen am anderen Ufer hinzufügen, erzeugen uns den Problemraum. Zustände, bei denen es mehr Kannibalen als Missionare gibt, werden nicht erzeugt. Das gewünschte Ergebnis ist der Pfad vom Ausgangszustand zum Zielzustand. Um die Operatoren etwas anschaulicher zu machen, seien einige in Prolog-Notation hier aufgeführt. Dabei geht die formale Repräsentation davon aus, daß nur ein Ufer explizit dargestellt werden muß. Ein Zustand wird dann repräsentiert durch die Anzahl der Missionare und die Anzahl der Kannibalen am Ausgangsufer sowie einen binären Wert, der gleich 0 ist, wenn das Boot nicht am Ausgangsufer ist, und gleich 1 ist, wenn das Boot dort ist. Der Anfangszustand ist dann

```
z(3,3,1)
```

- alle 3 Missionare und alle 3 Kannibalen und das Boot sind am Anfangsufer

Und der Zielzustand ist

```
z(0,0,X)
```

- kein Missionar und kein Kannibale sind am Anfangsufer. (Wir wissen auch, daß `X=0` sein wird, da das Boot nicht von selbst an das andere Ufer gelangt -- dies ist jedoch keine Bedingung.)

Die Operatoren geben die Fälle an, daß ein oder zwei Missionare vom Anfangsufer wegfahren oder dort ankommen, daß ein oder zwei Kannibalen vom Anfangsufer wegfahren oder dort ankommen, sowie daß ein Missionar und ein Kannibale vom Anfangsufer wegfahren oder ankommen. Es gibt also 10 spezielle Operatoren. Der Operator für die Überfahrt eines Missionars vom Ausgangsufer zum anderen könnte so aussehen:

```
o(z(M,K,1), z(Mneu, K, 0)):-
    M > 0,
    Mneu is M - 1,
    (K ≤ Mneu ; Mneu = 0),
    F1 is 3 - K,
    F2 is 3 - Mneu,
    (F1 ≤ F2 ; F2 = 0).
```

Der Operator für die Überfahrt von zwei Kannibalen zum Anfangsufer sieht so aus:

```
o(z(M,K,0), z(M, Kneu, 1)):-
    K < 3 - 1,
```

```
Kneu is K + 2,
(Kneu ≤ M ; M = 0),
F1 is 3 - Kneu,
F2 is 3 - M,
(F1 ≤ F2 ; F2 = 0).
```

Dabei ist das erste Argument des Operators der aktuelle Zustand und das zweite Argument der direkte Nachfolgezustand. Die erste Bedingung entscheidet über die Anwendbarkeit des Operators: es können nicht mehr Kannibalen oder Missionare wegfahren, als da sind. In der dritten und den folgenden Zeilen werden die Bedingungen dafür angegeben, daß kein Missionar verspeist wird. Zunächst wird es für das Ausgangsufer, dann komplementär für das andere Ufer abgeprüft. Das ";" bedeutet ein lokales ODER. "is" überträgt das Ergebnis der nachfolgenden Rechnung auf die vorn stehende Variable. Wenn alle Bedingungen erfüllt sind, ist der Folgezustand erzeugt. Ein Suchverfahren verwendet die Operatoren, um die Nachfolgezustände zu erzeugen.

3.1.1 Tiefensuche

Bei einer Tiefensuche wird der Graph so durchsucht, daß zunächst eine ganze Folge von Überfahrten bis zum Abbruch verfolgt wird . Es wird stets nur ein Pfad zur Zeit gespeichert und verfolgt. Um endlose Schleifen zu vermeiden, markiert man die Knoten, die bereits durchsucht worden sind. Obendrein muß man sich merken, welche Kante von diesem Knoten aus beschritten wurde, damit man sie nicht noch einmal wählt. Beim **Rückziehverfahren** (**backtracking**) wird der aktuelle Pfad markiert und gespeichert, so daß beim Abbruch an einem Knoten, der nicht die Lösung darstellt, zum letzten Knoten zurückgesprungen werden kann. Vom letzten Knoten aus wird dann eine andere Kante als die zuvor gewählte verfolgt.

Die folgende Prozedur führt eine Tiefensuche durch[5]:

1. Bilde eine einelementige Liste, die den Wurzelknoten mit dem Startzustand enthält.

2. Bis die Liste leer ist, nimm das erste Element der Liste.

   a) prüfe, ob das Element der Zielknoten ist;

      wenn ja, halte an und melde Erfolg; wenn nein, geht es weiter.

   b) wenn das Element Nachfolger hat, entferne es aus der Liste und setze seine Nachfolgeknoten als Elemente vorn in die Liste ein.

3. Wenn der Zielknoten gefunden wurde, melde Erfolg, sonst Mißerfolg.

In Prolog können wir dies folgendermaßen realisieren:

```
[library(basics)].      %damit ist member(X,Y) verfügbar, das
                        %prüft, ob X Element der Liste Y ist.

%%% tiefensuche (-bisheriger Pfad, +aktueller Knoten, -Gesamtpfad) %%%

tiefensuche(P, K, [K|P]):- ziel(K). %Abbruchbedingung Erfolg;
                                    %ziel(K) muß  definiert sein.
```

[5] Die einfachen Prozeduren für Tiefen-, Breitensuche und Bergsteigen sind an Pat Winstons Darstellung in Winston (1987) angelehnt.

```
tiefensuche(P, K, L):-
      nachf(K, K1),          % nachf ist der Operator, muß
                             % definiert sein
      \+ member(K1, P),      % \+ drückt die Negation aus,
                             % Zyklen vermeiden
      tiefensuche([K|P], K1, L).
```

3.1.2 Breitensuche

Die Breitensuche (breadth-first search) expandiert alle Knoten auf einer Ebene. Für alle so erreichten Knoten werden dann alle Nachfolgerknoten erzeugt. Die Breitensuche garantiert, daß die Lösung gefunden wird. Allerdings kann sie recht spät gefunden werden. Es ist also ein Suchverfahren, das wir anwenden, wenn wir nicht schätzen können, was vielversprechend ist, und was nicht.

Die folgende Prozedur führt eine Breitensuche durch:

1. Bilde eine einelementige Liste, die den Wurzelknoten mit dem Startzustand enthält.

2. Bis die Liste leer ist, nimm das erste Element der Liste.

   a) prüfe, ob das Element der Zielknoten ist;

      wenn ja, halte an und melde Erfolg; wenn nein, geht es weiter.

   b) wenn das Element Nachfolger hat, entferne es aus der Liste und setze die Nachfolger hinten in die Liste ein.

3. Wenn der Zielknoten gefunden wurde, melde Erfolg, sonst Mißerfolg.

Dadurch daß die Nachfolger hinten in der Liste gespeichert werden, statt wie bei der Tiefensuche vorn, treten alle Knoten einer Ebene in der Liste auf, statt wie bei der Tiefensuche nur diejenigen des aktuellen Pfads. Wie die Tiefensuche kann auch die Breitensuche durch eine lokale Abschätzung verbessert werden. Man expandiert dann schlecht bewertete Knoten gar nicht, führt also die Breitensuche nur mit den m besten Knoten einer Ebene fort.

Um die Breitensuche in Prolog aufzuschreiben, brauchen wir das Prädikat `findall/3`. Nehmen wir wieder an, wir hätten als Operatoren Klauseln mit dem Prädikat `nachf(+K,-L)`. Für einen bestimmten Knoten `a` liefert uns `findall/3` alle Nachfolger:

```
findall(X, nachf(a,X), Nachfolger).    %für alle X wende das Prädikat
                                       %nachf an und sammle dessen
                                       %Ergebnis in der Liste Nachfolger.
```

Jetzt können wir die Breitensuche einfach formulieren:

```
breitensuche([[K|P] | _], [K|P]):- ziel(K).  %Abbruchbedingung Erfolg
                                             %ziel(K) muß definiert
                                             %sein.

breitensuche( [[K|P] |Ps], L):-
  findall( [K1,K|P],
    (nachf(K,K1), \+ member(K1, [K|P]) ),    %nicht-zyklische Expansion
    Pneu),
  append(Ps, Pneu,P1), !,                    %Nachfolger hinten anhängen.
  breitensuche(P1, L).                       %rekursiver Aufruf.
```

3.1.3 Allgemeine Suche

Zwei Dinge steuern ein Suchverfahren: die Einsortierung der Nachfolgeknoten und die Informationen, die wir nutzen können, um Kosten für Wege abzuschätzen. Als allgemeines Suchverfahren, das eine globale Schätzfunktion benutzt, können wir die folgende Prozedur angeben[6]. Dabei sind in einer Liste OFFEN alle bisher erzeugten, aber nicht nicht expandierten Knoten enthalten. In der Liste GESCHLOSSEN sind alle bereits abgearbeiteten Knoten enthalten.

1. Die Liste OFFEN wird mit dem Anfangszustand initialisiert.

2. Die Liste GESCHLOSSEN wird als leere Liste initialisiert.

3. Falls OFFEN leer ist, bricht die Suche ab und es wird ausgegeben:

   "Zielzustand kann nicht erreicht werden."

4. Der erste Knoten n der Liste OFFEN wird aus OFFEN entfernt und der Liste GESCHLOSSEN hinzugefügt.

5. Wenn n der Zielzustand ist, bricht die Suche ab und der Pfad, der zu n führte, wird ausgegeben.

   Wenn n nicht der Zielzustand ist, wird n expandiert, so daß eine Menge M aller Nachfolger von n entsteht, die nicht schon Vorgänger von n sind. M wird als Nachfolger von n eingetragen.

6. Alle Elemente von M, die noch nicht in OFFEN oder GESCHLOSSEN vorhanden sind, werden der Liste OFFEN hinzugefügt.

7. Alle anderen Elemente aus M können dazu führen, daß Kanten umdirigiert werden müssen und dann auch die Kosten von schon erzeugten Nachfolge-Knoten aktualisiert werden müssen.

8. Die Liste OFFEN wird entsprechend der geschätzen Kosten sortiert, so daß der meistversprechende Knoten nach vorn kommt.

9. Rücksprung auf 3.

Wir können dies in Prolog so aufschreiben:

```
suche(Offen, Geschl, Ziel):-
      member(Ziel, Offen),          %Abbruchbedingung Erfolg.
      write(Geschl).

suche(Offen, Geschl, Ziel):-
      best(Offen, Best, RestOffen), %Sortierung von Offen; best
                                    %muß definiert sein.
      findall(Nachf,
              nachf(Best,Nachf),    % Nachfolger des besten
                                    % Knotens
              AlleNachf),
      verteile(AlleNachf, RestOffen, [Best|Geschl],NeuOffen),
                                    % keine Doppelten in
                                    % NeuOffen. verteile muß
```

[6] Die originale, englische Beschreibung von Suchverfahren ist in Nilsson (1980:64f) zu finden.

```
                                        % definiert sein.
      suche(NeuOffen, [Best|Geschl], Ziel). %rekursiver Aufruf

suche([], Geschl, Ziel):-          % Abbruchbedingung Mißerfolg.
      write('Misserfolg!'), !, fail.
```

Die Bestimmung des vielversprechendsten Knotens wird von der hier nicht angegebenen Klausel `best` geleistet. Wenn wir kein Kriterium finden, das uns hilft, nur vielversprechende Knoten zu expandieren, realisiert das Suchverfahren die Breitensuche, wenn die Kosten immer 1 sind. Da die Kosten des bisherigen Pfades für alle Expansionen eines Knotens gleich sind, werden dann alle von diesem Knoten fortführenden Kanten verfolgt. Das Verfahren realisiert eine uninformierte Tiefensuche, wenn die Liste OFFEN von `verteile` einfach so sortiert ist, daß die neuen Nachfolger nach vorn kommen, wobei `best` jeweils der erste Knoten wählt.

3.1.4 Bergsteigen

Die Reihenfolge, in der Kanten ausgewählt werden, ist entscheidend. Im idealen Fall würden wir nur einen Pfad verfolgen: einen, der zur Lösung führt! Wir können uns Kriterien überlegen, nach denen beurteilt wird, wie vielversprechend ein Knoten ist -- und schreiben dann `best` entsprechend. Mit anderen Worten, wir schätzen, ob ein Knoten zum Lösungspfad gehört, anhand bestimmter Kriterien. Da die Kriterien Annäherungen an Gesetzmäßigkeiten sind, wird eine Suche, die sich nach Kriterien richtet, heuristische Suche genannt. Ein solches Kriterium könnte die Unterschiedlichkeit des erreichten Zustands zum Zielzustand sein. Ein Verfahren, das nur den Abstand zum Ziel zu verringern sucht, heißt **Bergsteigen** (hill climbing). Tatsächlich führt so ein Kriterium zu einer sinnvollen Einschränkung der zu expandierenden Knoten. Es gibt aber Situationen, wo man sich vom Ziel wieder entfernen muß, um es schließlich zu erreichen! Solche Situationen kann eine derart lokale Abschätzung nicht behandeln. Es wird ja nur jeder einzelne Knoten mit dem Zielzustand verglichen. In dem Missionare-und-Kannibalen-Beispiel müssen zum Beispiel Personen wieder zurückfahren, damit am Ende alle am anderen Ufer sind, ohne daß ein Missionar gefressen würde. Es gibt Knoten, die bereits dichter am Ziel sind als der Nachfolgeknoten, der auf dem optimalen Pfad zum Ziel liegt. Ein solcher, sehr gut bewerteter Knoten heißt **lokales Maximum** und das Problem für einen Bergsteige-Algorithmus, von diesem nicht auf den richtigen Nachfolgeknoten zu kommen, heißt **Vorgebirgsproblem**.

Die folgende Prozedur führt Bergsteigen durch:

1. Bilde eine einelementige Liste, die den Wurzelknoten mit dem Startzustand enthält.

2. Bis die Liste leer ist, nimm das erste Element der Liste.

   a) prüfe, ob das Element der Zielknoten ist;

      wenn ja, halte an und melde Erfolg; wenn nein, geht es weiter.

   b) wenn das Element Nachfolger hat, entferne es aus der Liste. Ordne die Nachfolger nach dem geringsten Abstand zum Ziel an und füge den besten Nachfolger in die Liste ein. Die Liste enthält also nie mehr als ein Element.

3. Wenn der Zielknoten gefunden wurde, melde Erfolg, sonst Mißerfolg.

3.1.5 A *

Statt der lokalen Abschätzung können wir auch eine globale Abschätzung über den gesamten Pfad vornehmen. Wir können ein Kriterium suchen, das uns eine Abschätzung liefert, wie weit es von diesem Knoten zum Zielzustand ist, und ein weiteres Kriterium, das den zurückgelegten Weg vom Anfangsknoten zu diesem Knoten mißt oder abschätzt. Die globale Schätzfunktion für einen beliebigen Knoten n ist:

$$f(n) = g(n) + h(n)$$

wobei g(n) die Kosten des Pfades vom Anfangszustand zum Knoten n angibt und h(n) die Kosten vom Knoten n zum Zielzustand. An einem Knoten n kennen wir die Kosten des bisherigen Pfads, g(n). Die Kosten für den zukünftigen Pfad kennen wir nicht -- wir müssen sie schätzen. Im Gegensatz zum Bergsteigen vergleicht h(n) nicht nur den Knoten n mit dem Ziel, sondern schätzt den restlichen Pfad von n bis zum Ziel ab. Wenn wir die Kosten zu niedrig einschätzen, müssen möglicherweise sehr viele Knoten expandiert werden, bevor der minimale Pfad zur Lösung gefunden wird. Wenn wir die Kosten zu hoch einschätzen, kann der minimale Pfad übersehen werden - er befindet sich dann in dem nicht verfolgten Teil des Graphen. Wenn die Schätzfunktion h die untere Grenze der tatsächlichen Kosten h* trifft, so heißt das Suchverfahren, das sie nutzt, A*. Dieses Verfahren findet immer einen minimalen Pfad.

Nach Nilsson (1971) ist A* ein zulässiges Verfahren. Ein Suchverfahren ist **zulässig** (admissible), wenn es für jeden Graph einen optimalen Pfad findet und dann anhält, falls es einen solchen Pfad gibt. Der Beweis für die Zulässigkeit von A* läßt sich folgendermaßen nachvollziehen. Wir argumentieren für den Fall, daß es eine Lösung für das Suchproblem gibt. Wir schreiben die tatsächlichen Kosten für einen Pfad stets mit *, die geschätzten ohne Zusatz. Die tatsächlichen Kosten des Gesamtpfades werden als Kosten des Startknotens s notiert: f*(s). Für den Beweis der Zulässigkeit von A* betrachten wir zunächst das foldende Lemma.

Lemma: Wenn $h(n) \leq h^*(n)$ für alle Knoten n, dann gibt es immer für jeden optimalen Pfad zum Ziel einen Knoten n' dieses Pfades in OFFEN und es gilt $f(n') \leq f^*(n')$.

Beweis: Sei $n_0, n_1, ..., n_k$ ein optimaler Pfad vom Startknoten n_0 zum Zielknoten n_k. Sei n' der erste Knoten dieses Pfades in OFFEN. Es gibt immer einen solchen Knoten, denn sonst wäre n_k in GESCHLOSSEN und der Algorithmus terminiert. Da alle Vorgänger von n' bereits geschlossen sind, gilt $g(n')=g^*(n')$. Somit gilt auch

$$f(n') = g^*(n') + h(n') \leq g^*(n') + h^*(n') = f^*(n')$$

Theorem: Wenn $h(n) \leq h^*(n)$ und $cost(n) > \varepsilon > 0$ für alle Knoten n, dann ist A* zulässig.

Beweis: Angenommen A* terminiert nicht, indem es einen optimalen Pfad findet. Dann gibt es drei Fälle:

Fall 1:Was wäre, wenn A* anhielte, bevor das Ziel gefunden ist? Dann müßte OFFEN leer sein, denn sonst hält A* nicht an einem anderen als dem Zielknoten an. Von dem Lemma wissen wir aber, daß es immer einen Knoten n' in OFFEN gibt, der auf einem optimalen Pfad liegt, bevor A* anhält. Also kann A* nicht anhalten, bevor das Ziel gefunden wurde.

Fall 2: Was wäre, wenn A* nie anhält? Knoten, die mehr als

$$M = f^*(s)/\varepsilon$$

Schritte vom Start entfernt sind, werden nie geöffnet, denn sie wären teurer als irgendein Knoten n' in OFFEN, der auf einem optimalen Pfad liegt. Es gibt also nur endlich viele Knoten und Pfade zu diesen Knoten, die durchlaufen werden können, bis ein optimaler Pfad gefunden wird. Somit terminiert A* immer in endlicher Zeit.

Fall 3: Was wäre, wenn A* einen Lösungspfad finden, der nicht optimal ist? Ein solcher Pfad hätte dann ja höhere Kosten, d.h.

$$f^*(t) = g(t) > f^*(s).$$

Laut Lemma muß es vorm Anhalten einen Knoten n' in OFFEN gegeben haben, so daß

$$f(n') \leq f^*(s) \leq f(t).$$

Dann aber wählt A* den Knoten n' und nicht t.

Eine hinreichende Bedingung für $h(n) \leq h^*(n)$ (und somit für die Zulässigkeit) ist die **Monotonie** von h. Unter Monotonie wird verstanden, daß die geschätzten Kosten f für Nachfolger immer größer als die Kosten für Vorgängerknoten sind. Mit anderen Worten: Die Differenz der geschätzten Kosten für zwei aufeinanderfolgende Knoten ist nie größer als die tatsächlichen Kosten.

$$h(n_i) \leq h(n_j) + cost(n_i,n_j) \text{ wobei } n_j \text{ Nachfolger von } n_i \text{ ist.}$$

Zusätzlich zur Zulässigkeit von A* wollte Nilsson die Optimalität beweisen. Ein Verfahren A ist **informierter** als ein Verfahren B, wenn $h_A(n) > h_B(n)$ für alle inneren Knoten n gilt. Es wird angenommen, daß für alle Knoten dieselbe Schätzfunktion h verwendet wird. Nilsson argumentiert, daß A* **optimal** ist, insofern als es nie mehr Knoten expandiert als irgendein anderes zulässiges Verfahren, das weniger informiert ist als A*. Was wäre, wenn A* Knoten öffnen würde, die A nicht öffnet? Da A zulässig ist, muß A die Kosten solcher Knoten für teurer als $f^*(s)$ halten, denn sonst würden sie geöffnet. Also setzt A $h(n) \geq f^*(s) - g^*(n)$. Dann wäre aber A informierter als A*, was wir anders vorausgesetzt haben. Also ist A* optimal. Gegen diese Argumentation wendet sich Gelperin (1977). Hier sei von der Diskussion nur angeführt, daß A ja Information besser nutzen könnte, z.B. indem es die Liste GESCHLOSSEN analysiert oder indem es eine Schätzfunktion f besitzt, die nicht lediglich g und h addiert.

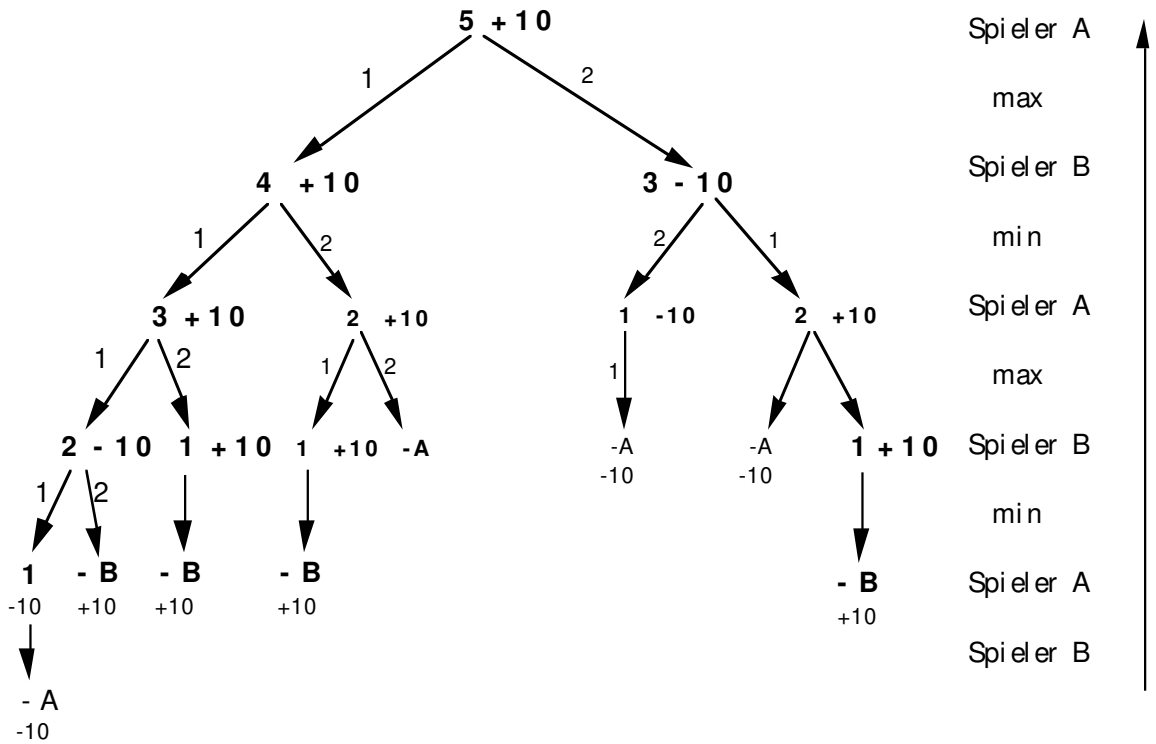
3.1.6 Spielbäume

Der Unterschied zwischen den eben betrachteten Graphen und Spielbäumen ist nicht nur, daß jeder Knoten nur einen Vorgängerknoten hat, also ein Baum ist. Wesentlich ist, daß für die Knoten einer Ebene[7] der eine Spieler die Nachfolgerknoten bestimmt, während für die Knoten der nächsten Ebene der andere Spieler die Nachfolger bestimmt. Man geht bei Spielbäumen also von zwei gegeneinander spielenden Personen aus. Wenn der Vorteil des einen Spielers genau dem Nachteil

[7] Die Ebene wird Halbzug (ply) genannt. Die Anzahl der Halbzüge ist die Tiefe des Baums.

des anderen Spielers entspricht, spricht man von einem Nullsummenspiel, weil sich Nachteile (dargestellt durch negative Zahlen) und Vorteile (dargestellt durch positive Zahlen) beider Spieler genau zu 0 addieren. In einer solchen Situation gibt es nicht einen Zielzustand und einen optimalen Pfad, sondern für jeden Spieler einen anderen. Die Blätter oder Endpunkte des Baumes geben Gewinn oder Verlust für jeden Spieler an. Diese Information soll möglichst schon am Anfang verwendet werden. Dabei kann jeder Spieler davon ausgehen, daß im für ihn schlimmsten Fall der Gegenspieler, wenn er an der Reihe ist, den für ihn selbst schlechtesten, für den Gegenspieler aber besten Nachfolgezustand wählen wird.

Ein einfaches Beispiel ist "den letzten beißen die Hunde": ein Stapel von 5 Karten liegt auf dem Tisch, von dem jeder Spieler abwechselnd entweder eine oder zwei abnimmt. Wer die letzte Karte des Stapels nimmt, hat verloren. Der Spielbaum ist im folgenden so notiert, daß die Anzahl der genommenen Karten an den Kanten, die Anzahl der auf dem Stapel verbleibenden an den Knoten und der Verlust als "-" notiert ist.



Vom Ergebnis aus kann A den Baum analysieren und danach den ersten Zug entscheiden. Links unten ist ein Zustand, in dem A verloren hat. Wir können diesem Zustand eine Bewertung zuordnen, die negativ ist, z.B. -10. Der Zustand läßt sich nicht vermeiden, wenn der Vorgängerzustand erreicht ist. Also kann die negative Bewertung an den Vorgängerzustand hochgereicht werden. Der Vorgängerzustand läßt sich von A nicht vermeiden, wenn 2 Karten vorhanden sind und B am Zug ist. Dann wird B nicht freiwillig verlieren, indem B zwei Karten nimmt. Das wäre eine positive Bewertung für A, die wir mit +10 notiert haben. B wählt aber immer negativ bewertete Zustände. Also bewertet A schon diesen Zustand mit -10 und nimmt in der Situation, wo drei Karten auf dem Stapel liegen, zwei davon. Dann kann B nur noch verlieren, der Zustand bekommt also vom Verlust von B die positive Bewertung hochgereicht. Das wird wiederum B nicht zulassen und deshalb in der Situation mit 4 Karten nicht eine Karte nehmen. B sucht nach einem Zustand, der niedriger bewertet ist als +10. Leider führt aber das Nehmen von zwei Karten ebenfalls zu Bs Verlust. Es gibt keinen schlechter bewerteten Nachfolgezu-

stand. Deshalb wird bereits die Situation mit 4 Karten auf dem Stapel mit +10 positiv bewertet. Da A anfängt, wird A sicherlich den positiv bewerteten Zustand wählen.

Die Bewertungen der Ergebnisse werden also nach oben weitergereicht. Die tatsächlichen Ergebnisse sind an den Blättern des Baumes aufgezeichnet. Sie bilden die Suchgrenze. Wenn A am Zug ist, erhält der Knoten den Wert des am besten bewerteten Nachfolgeknoten. Das ist die max-Ebene. Wenn B am Zug ist, erhält der Knoten den Wert des am schlechtesten bewerteten Nachfolgers. Das ist die min-Ebene. Denn A wählt den für A besten, B den für A schlechtesten Zug. Dieses Verfahren, Bewertungen von unteren Knoten an obere hochzureichen, heißt **Minimax-Algorithmus**, weil abwechselnd maximale und minimale Werte nach oben übertragen werden. In dieser Weise werden die Werte bis zum Anfangszustand hochgereicht. Wenn wir den gesamten Baum durchgehen können, wissen wir bereits am Anfang, wer gewinnt.

Wir können eine Prozedur für das Minimax-Verfahren angeben[8]:

1. Bestimme, ob die aktuelle Ebene Suchgrenze, eine min-Ebene oder eine max-Ebene ist.

2. Wenn die Suchgrenze erreicht wurde, gib die statische Bewertung als Ergebnis bekannt!

3. Wenn es eine min-Ebene ist, verwende die Minimax-Prozedur bei den Nachfolgern der gegenwärtigen Stellung und gib das Minimum der Ergebnisse an!

4. Wenn es eine max-Ebene ist, verwende die Minimax-Prozedur bei den Nachfolgern der gegenwärtigen Stellung und gib das Maximum der Ergebnisse an!

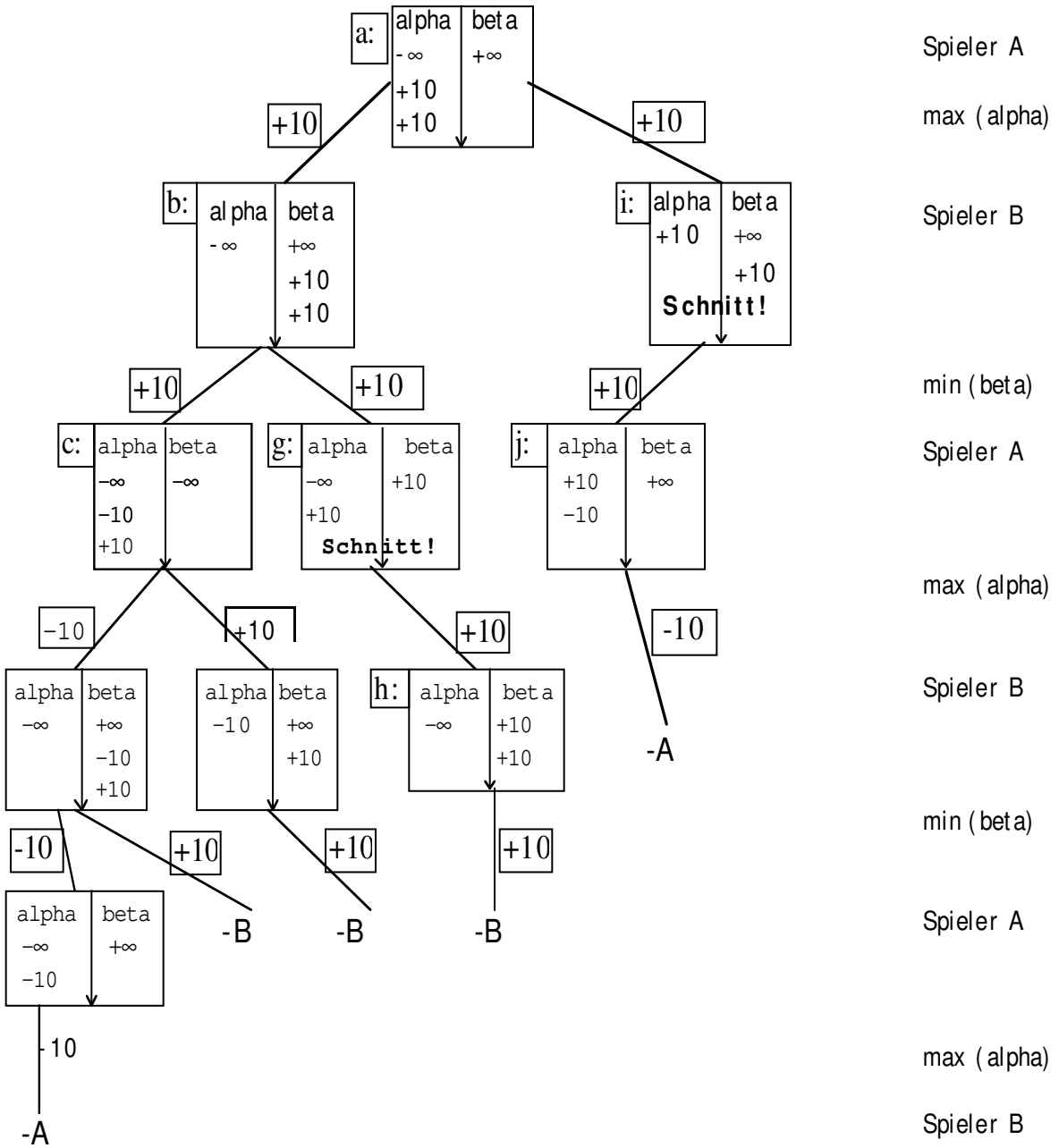
Dieses Verfahren setzt voraus, daß wir den gesamten Baum erzeugen oder wenigstens bis zu einer Tiefe, bei der eine Bewertung der Stellungen möglich ist (Suchgrenze). Es handelt sich um eine Tiefensuche mit einer Bewertungsfunktion, deren Bewertungen zurück nach oben übertragen werden. Der Baum des Beispiels hat im schlimmsten Falle eine Tiefe d von 5 Halbzügen und eine Verzweigung b an jedem Knoten (Anzahl der Nachfolger) von 2. Die Anzahl der Blätter ist maximal b^d, also im Beispiel 32. Die Menge der Blätter wächst exponentiell mit der Baumtiefe. Die Anzahl aller explorierten Knoten ist schlimmstenfalls $b^{d+1}-1$, im Beispiel also 63. Wir können den Baum aber beschneiden.

Tatsächlich habe ich den rechten Zweig des Baumes nicht mehr beschrieben, weil der Zustand mit 4 Karten auf dem Stapel bereits die höchste Bewertung hat - warum sollte A dann über den anderen Zweig noch nachdenken, der nicht besser sein kann! A kann aufhören, wenn eine neue Bewertung nicht besser wird als die bisher schlechteste; B kann aufhören, wenn eine neue Bewertung nicht kleiner wird als die bisher größte Bewertung. Wir können also ein beschränkendes Intervall einführen, dessen untere Grenze **alpha** der minimale Wert ist, den A garantiert schon erreicht hat, und dessen obere Grenze **beta** der maximale Wert ist, den A erhoffen kann zu erreichen. Aus der Sicht von B ist beta der minimale Wert, den B bestimmt erreichen kann. Jeder Knoten, dessen Bewertung außerhalb dieses Intervalls liegt, ist irrelevant. Die Suche wird abgebrochen,

8 Die Darstellung entspricht der von Winston (1987).

- wenn eine Bewertung eines Knoten, an dem A am Zuge ist, kleiner oder gleich alpha ist, oder
- wenn eine Bewertung eines Knoten, an dem B am Zuge ist, größer oder gleich beta ist.

Anders ausgedrückt, es wird nur weitergesucht, wenn A einen höheren Wert als alpha erreichen kann, oder wenn B einen kleineren Wert als beta erreichen kann[9]. Ein Verfahren, das den Minimax-Algorithmus um das Beschneiden mithilfe von alpha und beta erweitert, heißt **Alpha-Beta-Algorithmus**.



[9] Wie man am Beispiel sehen kann, ist ein kleinerer Wert gut für B, denn Bs Verlust wird mit +10 notiert, Bs Gewinn mit -10!

In jedem Knoten kann die Entwicklung des alpha- und beta-Wertes nachvollzogen werden. Auf der alpha-Seite ist jeweils über den aktuellen Werten zu maximieren, auf der beta-Seite zu minimieren. Alpha und beta sind lokale Variablen in jedem Knoten, die jeweils nur von einem Vorgängerknoten an seine Nachfolger weitergereicht werden.

Im Beispiel wird zunächst gemäß der Tiefensuche der linkeste Teilbaum aufgebaut. Der linkeste unterste Knoten erhält die Bewertung -10, die hochgereicht wird und als neuer alpha-Wert eingetragen wird. Der Knoten darüber erhält von seinem zweiten Unterknoten den Wert +10 hochgereicht, der für beta eingetragen wird. Da es sich aber um ein minimierendes Hochreichen handelt, wird -10 an den Knoten darüber weitergegeben. Hier, bei Knoten c, haben wir nun auch noch vom Schwesterknoten den Wert +10. Dies ist der höchste Wert, den A bekommen kann, also beta. Es wird dann der zweitlinkeste Pfad exploriert, bei dem 2 Karten auf dem Stapel liegen (Knoten g). Dieser Knoten wird zunächst so expandiert, daß A eine Karte nimmt, was dazu führt, daß B die letzte Karte nehmen muß und verliert. Der Knoten h mit einer Karte erhält als aktuelle Bewertung +10. Die zweite Expansion des Knoten g mit den 2 Karten muß nun nicht mehr untersucht werden, denn die Bewertung von h ist bereits +10, also größer oder gleich beta. Nach der beta-Regel schneiden wir also alle weiteren Expansionen ab. Dann wird der rechte Teilbaum exploriert. Knoten i wird expandiert, so daß der erste Nachfolgerknoten j eine Bewertung erhalten kann, nämlich -10. Diese Bewertung ist nicht größer als der schlechteste Wert, mit dem A rechnen kann. Die Bewertung von j ist kleiner gleich alpha. Nach der alpha-Regel brauchen wir keine weiteren Expansionen des Knoten i zu untersuchen. Der gesamte so beschnittene Baum hat nur noch 5 Blätter. Dies entspricht ungefähr der Quadratwurzel aus der Anzahl der maximalen Blätter. Es ist etwas weniger, weil schon der unbeschnittene Baum weniger als 32 Blätter hatte.

Wir können eine Prozedur für das Alpha-Beta-Verfahren angeben[10]:

Bestimme, ob die Ebene die oberste Ebene ist, oder ob die Suchgrenze erreicht wurde, oder ob es eine min-Ebene oder eine max-Ebene ist.

a) Falls es die oberste Ebene ist, soll alpha $-\infty$ und beta $+\infty$ sein.

b) Falls die Suchgrenze erreicht wurde, berechne den statischen Wert der Stellung und gib ihn als Ergebnis zurück.

c) Falls es eine min-Ebene ist, führe die folgenden Schritte durch, bis alle Nachfolger durch Minimax geprüft sind oder alpha $\geq$ beta:

c1) Beta wird das Minimum von: dem beta-Wert des Elternknotens und dem kleinsten bisher durch die Minimax der Nachfolger ermittelten Wert.

c2) Wende Minimax auf den nächsten Nachfolger der gegenwärtigen Stellung an und ersetze das vorliegende alpha und beta durch die Ergebnisse dieser neuerlichen Anwendung von Minimax.

c3) Liefere beta als Ergebnis.

d) Falls es eine max-Ebene ist, führe die folgenden Schritte durch, bis alle Nachfolger durch Minimax geprüft sind oder alpha $\geq$ beta:

[10] Die Darstellung entspricht der von Winston (1987).

d1) Alpha wird das Maximum von: dem alpha-Wert des Elternknotens und dem größten bisher durch Minimax der Nachfolger ermittelten Wert.

d2) Wende Minimax auf den nächsten Nachfolger der gegenwärtigen Stellung an und ersetze das vorliegende alpha und beta durch die Ergebnisse dieser neuerlichen Anwendung von Minimax.

d3) Liefere alpha als Ergebnis!

Im allgemeinen Fall wird die Verzweigung b eines Baumes durch den Alpha-Beta-Algorithmus verringert, und zwar maximal auf die Quadratwurzel von b. Im Beispiel verringert sich die Verzweigung von 2 auf die Wurzel von 2, also 1,4. Insgesamt brauchen bei gerader Tiefe (dann ist B als letzter am Zug) nur $2b^{d/2} - 1$ Blätter statisch bewertet zu werden. In unserem Beispiel sind das 7 Blätter. Bei ungerader Tiefe (A ist als letzter am Zug) werden bestenfalls nur $b^{(d+1)/2} + b^{(d-1)/2}$ Blätter bewertet. In unserem Beispiel sind das 4,28. Im besten Falle kann der Alpha-Beta-Algorithmus doppelt so tief suchen wie ein erschöpfender Minimax-Algorithmus. Im schlechtesten Fall expandiert auch der Alpha-Beta-Algorithmus alle Knoten.

Unser Beispiel hat nur zwei Bewertungen: +10 und -10. Es kann aber auch Zwischenbewertungen geben wie z.B. -5, 0, +5. In diesem Falle sind die vom Alpha-Beta-Algorithmus hochgereichten Knoten-Bewertungen nicht exakt. Bei Zwischenwerten kann die korrekte Bewertung für einen Knoten, bei dem A am Zug ist, größer sein als vom Alpha-Beta-Algorithmus errechnet. Die Bewertung für einen Knoten, bei dem B am Zug ist, kann kleiner sein als vom Alpha-Beta-Algorithmus angegeben. Wir wissen aber, daß diese Unkorrektheit für das Spielen nicht wichtig ist: A muß sich nicht überlegen, was sein oder Bs schlechtest möglicher Zug wäre.

Bei realistischen Spielen wie z.B. Schach kann der Wert für alpha und beta nicht dadurch gefunden werden, daß man ein Endergebnis nach oben reicht. Die Suche wird in der Tiefe beschränkt, z.B. auf 7 Halbzüge. Die Bewertung der Knoten auf der Ebene 7 muß dann durch eine Bewertungsfunktion gefunden werden. Diese kann die Erfolgsaussichten für das Gewinnen oder Verlieren abschätzen. Dazu werden beim Schach etwa die Wertigkeit der Figuren und bestimmte Stellungsmerkmale (z.B. Läufer am Rand) genutzt. Diese geschätzten Bewertungen werden dann mit dem Minimax-Algorithmus nach oben gereicht, so daß alpha und beta bestimmt werden können. Die Tiefenbeschränkung kann auch iterativ angewandt werden: zunächst wird der Alpha-Beta-Algorithmus für eine bestimmte Tiefe angewandt und dann wird die Liste der Nachfolgezustände gemäß der Bewertung sortiert und wieder, für eine größere Tiefe, der Alpha-Beta-Algorithmus angewandt. Dies wird fortgesetzt bis eine bestimmte Zeitgrenze erreicht ist. Die zu dem Zeitpunkt aktuellen Bewertungen werden für die Auswahl des nächsten Zuges genutzt.

Bei dem - auch psychologisch untersuchten - Schachspiel wird besonders deutlich, daß die Formalisierung eines Spiels als Suchproblem zwar die Aufgabe beschreibt, nicht jedoch das menschliche Vorgehen. Je besser ein Schachspieler ist, desto weniger Stellungen überlegt er sich überhaupt. Einige 10 Stellungen werden sehr genau untersucht, alle anderen werden gar nicht betrachtet (De Groot 1965). Die erfolgreichen Schachprogramme, die mit den oben beschriebenen Verfahren arbeiten und bereits Weltmeister schlagen, untersuchen Millionen (und mehr) Stellungen. Je größer die Suchtiefe, desto erfolgreicher spielen die Programme. Allerdings verwenden sie für den Eröffnungsteil der Schachpartien auch menschliches Wissen, das ihnen in Form einer Eröffnungsbibliothek zur Verfügung gestellt wird.

3.1.7 Literatur

Barr, A., Davidson, J. (1979): Representation of Knowledge, in: Barr, Feigenbaum (eds): *The Handbook of AI*,

De Groot, A.D. (1965): Thought and Choice in Chess, The Hague: Mouton

Gelperin, D. (1977): On the Optimality of A*, in: Artificial Intelligence Journal, 8, S. 69-76

Nilsson,N. J. (1980): Principles of Artificial Intelligence, Berlin: Springer

**Nilsson,N. J. (1971): Problem-Solving Methods in Artificial Intelligence, McGraw-Hill

Pearl, J. (1984): Heuristics - Intelligent Search Strategies for Computer Problem Solving, Reading, MA: Addison-Wesley

**Winston, P. (1987): Künstliche Intelligenz, Bonn: Addison-Wesley.

3.2 Problemlösung als Beweis

Die Beschreibung von Problemen in einer Form, die die Lösung des Problems erlaubt, ohne daß man das Problem aus der sinnlichen Wahrnehmung heraus begreifen muß, ist der Jahrhunderte alte Traum von Logikern. Schon an der Form einer Aussage soll man erkennen können, ob sie wahr ist oder nicht. Wenn zwei Formen verschieden aussehen, obwohl sie doch dasselbe bedeuten, soll man mithilfe von einfachen Manipulationen erkennen können, daß sie "eigentlich" gleich sind. Diese Manipulationen verändern nicht den Wahrheitsgehalt, sie sind **wahrheitserhaltend** und **falschheitserhaltend**. Die Formen bestehen aus verabredeten (per Konvention festgelegten) Zeichen (das sind Symbole) und den Möglichkeiten ihrer Anordnung. Dies ist die **Syntax**. "Syntaxis" ist das griechische Wort für Anordnung.

Die **Bedeutung** von Aussagen, die in einer bestimmten Anordnung von Symbolen dargestellt sind, ist ihr Wahrheitswert. Die Bedeutung eines Bezeichners (z.B. "Baum" oder "Uta") sind die bezeichneten Objekte selbst (z.B. alle Bäume oder eine Person). Es gibt Aussagen, die keinen Wahrheitswert haben. So sind Hoffnungen oder Befürchtungen nicht wahr oder falsch, sondern vielleicht berechtigt, (un-)angenehm, anspornend. Der Satz "Hoffentlich komme ich heute nicht zu spät." hat in diesem Sinne keine Bedeutung. Aber natürlich ist es ein sinnvoller Satz. Wir trennen **Sinn** von Bedeutung seit Gottlob Frege. Der Sinn eines Satzes ist unabhängig von seinem Wahrheitsgehalt. Wir verstehen den Satz "Der Stuhl ist rot" unabhängig davon, ob er wirklich rot ist. Der Sinn eines Bezeichners hängt mit unserer Wahrnehmung zusammen. Es ist seine Verankerung in unserer Erfahrung. Der Sinn kann verschieden sein, wo die Bedeutung gleich ist. Eigentlich umfaßt **Semantik** sowohl Sinn als auch Bedeutung. In der Logik befaßt man sich meist mit der Bedeutung, wobei man sie allerdings im Sinn zu verankern sucht.

Im folgenden wird eine kurze Einführung in die Logik gegeben. Dabei behandeln wir hier nur die Aussagenlogik. Die Aussagenlogik behandelt nicht die einzelnen Bestandteile von Aussagen wie etwa Quantoren (alle, einige, mindestens eine, genau eine, viele...), Individuen, Variablen. Stattdessen wird ein Sachverhalt durch einen einzigen Ausdruck zusammengefaßt. Diesem Ausdruck wird ein Wahrheitswert zugeordnet.

3.2.1 Syntax

Die Aussagenlogik notiert komplette Sachverhalte durch Aussagensymbole. So ist zum Beispiel der_Stuhl_ist_rot eine Aussagensymbol. Geläufiger ist allerdings A, B, ... Aussagensymbole sind atomare Formeln oder Atome. Sie können mit Verknüpfungszeichen zu weiteren Formeln zusammengesetzt werden. Die **Syntax** der Aussagenlogik definieren wir induktiv:

1) Die Wahrheitswerte und Aussagensymbole sind Formeln.

2) Sind A und B Formeln, so auch

   $\neg A$, $A \,\&\, B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$

3) Das sind alle Formeln.

Dirk Siefkes hat in seinem schönen Buch "Formalisieren und Beweisen" (1990)[11] das folgende Beispiel in Aussagenlogik notiert: Von vier Kindern hat eines einen Ball in ein Fenster geworfen. Anne sagt: "Emil war's." Emil sagt:"Nein, Gustaf." Gustaf sagt:"Emil lügt." Fritz sagt:" Ich war's nicht!" Nur ein Kind sagt die Wahrheit, alle anderen lügen. Für "Anne hat den Ball ins Fenster geworfen" schreiben wir `Aball`, entsprechend `Eball`, `Gball`, `Fball` für die anderen möglichen Werfer. Für "Anne hat die Wahrheit gesagt" schreiben wir `Awahr`, entsprechend `Ewahr`, `Gwahr`, `Fwahr` für die anderen möglichen (Nicht)Lügner. Der ganze Fall kann also folgendermaßen in Aussagenlogik geschrieben werden:

`Awahr ↔ Eball`

`Ewahr ↔ Gball`

`Fwahr ↔ ¬ Fball`

`Gwahr ↔ ¬ Ewahr`

`Awahr →¬ Ewahr & ¬Fwahr & ¬ Gwahr`

`Ewahr → ¬ Awahr & ¬ Fwahr & ¬ Gwahr` nur ein Kind sagt die Wahrheit!

`Fwahr → ¬ Awahr & ¬ Ewahr & ¬ Gwahr`

`Gwahr → ¬ Awahr & ¬ Ewahr & ¬ Fwahr`

`Awahr ∨ Ewahr ∨ Fwahr ∨ Gwahr` eins der Kinder warf den Ball

`Aball → ¬ Eball & ¬ Fball & ¬ Gball`

`Eball → ¬ Aball & ¬ Fball & ¬ Gball` nur ein Kind warf den Ball!

`Fball → ¬ Aball & ¬ Eball & ¬ Gball`

`Gball → ¬ Aball & ¬ Eball & ¬ Fball`

Um mit diesen Aussagen arbeiten zu können, müssen wir aber auch Wahrheitswerte zuordnen können. Wir wollen aus diesen Formen ja ablesen können, ob es möglich ist, daß genau ein Kind die Wahrheit sagt und genau ein Kind den Ball ins Fenster geworfen hat, und - wenn ja - welches Kind es war.

3.2.2 Semantik

Die Wahrheitswerte sind nur wahr (`W`) und falsch (`F`). Für Verknüpfungen gibt es Wahrheitstafeln, die die Wahrheitsfunktion (Boolesche Funktion) von Wahrheitswerten in Wahrheitswerte angeben:

[11] Dieser Abschnitt ist eng an dies Buch angelehnt.

¬	W	F
	F	W

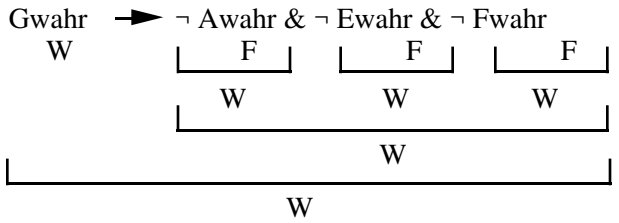
&	W	F
W	W	F
F	F	F

◊	W	F
W	W	W
F	W	F

→	W	F
W	W	F
F	W	W

↔	W	F
W	W	F
F	F	W

Da wir bei den Aussagen nicht von vornherein wissen, ob sie wahr oder falsch sind, belegen wir sie mit den möglichen Wahrheitswerten. Eine **Belegung** einer Formel ist ein Abbildung von der Menge der Aussagensymbole der Formel in die Menge der Wahrheitswerte. Wir werten eine Formel unter einer Belegung aus, indem wir jedes Aussagensymbol durch seinen Wahrheitswert unter der Belegung ersetzen und dann die Wahrheitstafeln anwenden. Eine Formelmenge X ist wahr unter einer Belegung, wenn jede Formel aus X unter der Belegung wahr ist. Die Formelmenge ist also wie eine Konjunktion von Formeln. Die Auswertung der Formel `Gwahr` → ¬ `Awahr & ¬ Ewahr & ¬ Fwahr` für eine willkürlich gewählte Belegung ergibt:



Wir können natürlich auch umgekehrt fragen, unter welchen Belegungen die Formel wahr wäre. Die Formel wäre wahr, wenn `Gwahr` falsch wäre und mindestens eins von `Awahr, Ewahr, Fwahr` wahr ist; oder wenn `Gwahr` wahr ist und `Awahr, Ewahr, Fwahr` alle falsch sind. Wenn alle Formeln des Falls wahr sein sollen und wenn `Gwahr` wahr sein soll, sind die möglichen Belegungen eingeschränkt. Wir gehen die Formeln des Fenster-Falles durch und rechnen die Belegungen rückwärts: `Awahr` ist bereits mit `falsch` belegt. Damit die erste Formel wahr wird, muß `Eball` mit `falsch` belegt werden. `Ewahr` ist schon mit falsch belegt; damit die zweite Formel wahr wird, wird `Gball` mit falsch belegt. `Fwahr` ist bereits mit `falsch` belegt. Damit die dritte Formel wahr wird, muß ¬`Fball` falsch sein, also `Fball` wahr. Damit ist das Problem gelöst, wer den Ball durch das Fenster geworfen hat. Alle weiteren Formeln lassen sich mit Wahrheitswerten belegen.

Mit einem ähnlichen Vorgehen läßt sich auch die Frage beantworten, ob Anne die Wahrheit gesagt hat. Wenn alle Formeln wahr sind und `Awahr` wahr ist, dann sind `Ewahr, Fwahr, Gwahr` falsch. Wenn aber `Gwahr` falsch ist und die vierte Formel soll wahr sein, dann muß `Ewahr` wahr sein. Das ist ein Widerspruch! Es gibt also keine Belegung, so daß `Awahr` wahr ist und alle Formeln wahr sind. Wenn `Awahr` wahr ist, dann ist die Formel

`Gwahr` ↔ ¬ `Ewahr`

falsch. Anne ist mit einem Widerspruchsbeweis überführt.

3.2.3 Erfüllbarkeit, Allgemeingültigkeit, Folgerbarkeit

Eine Formel oder Formelmenge heißt **erfüllbar**, wenn es eine Belegung gibt, unter der sie wahr ist. So war eben im Beispiel die Formelmenge erfüllbar, weil mit der Belegung `wahr` für `Gwahr` verträgliche Belegungen für alle anderen Aussagensymbole gefunden werden konnten.

Die Formeln des Fenster-Falles sind nicht **allgemeingültig**, weil es eine Belegung gibt, bei der nicht alle Formeln wahr sind. Allgemeingültig ist eine Formel oder Formelmenge genau dann, wenn sie unter jeder Belegung wahr ist (sind).

Daß Fritz den Ball geworfen hat, wurde deutlich unter der Annahme, daß alle Formeln wahr sein sollen und Gustaf die Wahrheit gesagt hat. Wir brauchen die zweite Annahme aber gar nicht. Tatsächlich ist es so, daß immer, wenn alle Formeln wahr sind, auch `Gwahr` und `Fball` wahr sind. Gwahr und Fball sind aus der Formelmenge (logisch) **folgerbar**. Eine Formel A folgt logisch aus einer Formel B (bzw. Formelmenge)

`B |= A (bzw. X |= A),`

wenn unter jeder Belegung, unter der B bzw. X wahr ist, auch A wahr ist. Eine Formelmenge Y folgt aus X, wenn jede Formel von Y aus X folgt. Die logische Folgerung ist

transitiv: `X |= Y, Y |= Z, also X |= Z,`

reflexiv: `X |= X`

> Auch die folgende, allgemeinere Beziehung gilt: $Y \supseteq X$, also Y |= X, d.h. Formeln folgen logisch aus sich selbst und weiteren Formeln.

Aus mehr Formeln kann man mindestens gleichviel folgern:

> $Y \supseteq X$, `X |= Z, also Y |= Z.`

Die Folgerung `A |= B` ist auf der Metaebene wie $A \rightarrow B$ auf der Ebene darunter. Wenn $A \rightarrow B$ wahr werden soll, muß man eine geeignete Belegung für `A` und `B` finden. Wenn `A |= B` wahr werden soll, muß es für alle Belegungen von `A` und `B` so sein. Also ist `A |= B` gleichbedeutend damit, daß $A \rightarrow B$ allgemeingültig ist.

Wenn wir zeigen wollen, daß aus unseren Fenster-Formeln logisch folgt, daß Gustaf die Wahrheit gesagt hat und Fritz den Ball geworfen hat, müssen wir also zeigen, daß alle Belegungen, die die Formeln wahr machen, auch `Gwahr` und `Fball` wahr machen. Woher wissen wir denn, daß wir die Annahme, daß `Gwahr` wahr ist, gar nicht hinzuzunehmen brauchen, um Fritz zu überführen? Vielleicht gibt es noch viel mehr Belegungen, unter denen die Formelmenge wahr ist und nicht immer ist dann auch `Gwahr` und `Fball` wahr. Wie können wir **entscheiden**, ob `Gwahr` und `Fball` aus den Formeln folgen? Ist das überhaupt entscheidbar? Dies ist das Problem der **Entscheidbarkeit** für die logische Folgerung. Das Problem ist für die Aussagenlogik zu lösen: man braucht nur alle Belegungen miteinander zu kombinieren für alle Aussagensymbole - und das sind ja endlich viele. In Dirk Siefkes Beispiel sind es 8. Bei zwei Wahrheitswerten haben wir also 2^8 Möglichkeiten zu prüfen, ob, immer wenn die Formeln wahr sind, auch `Gwahr` und `Fball` wahr ist. Da man bereits bei kleinen Beispielen sehr viel mehr Aussagensymbole braucht, hilft die Entscheidbarkeit nicht so viel.

Eine andere Methode, um zu zeigen, daß Fritz den Ball geworfen haben muß, ist der Widerspruchsbeweis. Um zu zeigen, daß `Fball` wahr sein muß, nimmt man an, daß `Fball` falsch sei. Wenn das zu einem Widerspruch führt, heißt das, daß `Fball` wahr sein muß, um die Formeln zu erfüllen. Daß man dies tun darf, kann man mit wahrheitserhaltenden Manipulationen beweisen. Man kann zeigen, daß A |= B, indem man zeigt, daß $A \rightarrow B$ allgemeingültig ist (s.o.). Jetzt ist die Frage, ob dies mit $A \,\&\, \neg B \rightarrow F$ gleichbedeutend ist. Dabei wenden wir nur Manipulationen an, die unabhängig von den Belegungen von `A` und `B` sind:

$(A \rightarrow B \vee C) \leftrightarrow (A \& \neg B \rightarrow C)$

Wir nehmen als C die Formel F (für falsch): $A \& \neg B \rightarrow F$

Der Widerspruchsbeweis ist also zulässig. Er setzt allerdings voraus, daß die Formeln nicht ohnehin zu einem Widerspruch führen.

Eine Formelmenge X heißt **widersprüchlich**, wenn es eine Formel A gibt, so daß A und $\neg A$ logisch aus X folgen. Sonst heißt X widerspruchsfrei. Eine Formelmenge ist widersprüchlich genau dann, wenn $X \models F$. Eine widersprüchliche Menge von Formeln ist nicht erfüllbar. Und aus einer widersprüchlichen Formelmenge folgt jede beliebige Formel. Also wieder:

$X \models A$ genau dann, wenn $X \& \neg A$ widersprüchlich ist.

Schließlich sei noch der Beweis durch **Kontraposition** angeführt. $A \models B$ wird bewiesen durch $\neg B \models \neg A$. Das ist zulässig, weil $A \models B$ mit $A \rightarrow B$ gezeigt werden kann, was äquivalent ist mit $\neg B \rightarrow \neg A$.

3.2.4 Formen

Wenn es ein Ziel der Logik ist, an der äußeren Form die Bedeutung abzulesen, ist eine Normalform nützlich, bei der Gleiches immer gleich aussieht und in der sich alles ausdrücken läßt. Die wahrheitserhaltenden Manipulationen sind dann dazu nötig, in diese Form zu überführen.

Ein **Literal** ist ein positives Atom (P) oder ein negatives Atom ($\neg P$). Die doppelte Negation ergibt einfach ein positives Atom (P statt $\neg\neg P$).

Die **konjunktive Normalform**[12] besteht aus Formeln, die Wahrheitswerte sind oder ein Konjunktion von Disjunktionen von Literalen.

$$F,\ W,\ (A_{1,1} \vee \ldots \vee A_{1,n}) \ \& \ \ldots \ \& (A_{m,1} \vee \ldots \vee A_{m,o})$$

In keiner Disjunktion darf ein Atom mehrfach vorkommen, auch nicht negiert.

Zum Umformen in diese Form braucht man vor allem die folgenden Manipulationen:

$A \leftrightarrow B$ wird zu $(A \rightarrow B) \& (B \rightarrow A)$

$A \rightarrow B$ wird zu $\neg A \vee B$

$\neg(A \& B)$ wird zu $\neg A \vee \neg B$, $\neg(A \vee B)$ wird zu $\neg A \& \neg B$

$\neg W$ wird zu F, $W \vee A$ wird zu W, $W \& A$ wird zu A,

$\neg F$ wird zu W, $F \vee A$ wird zu A, $F \& A$ wird zu F,

$A \vee (B \& C)$ wird zu $(A \vee B) \& (A \vee C)$.

Man schreibt $(A_{1,1} \vee \ldots \vee A_{1,n})$ auch als Menge $\{A_{1,1}, \ldots, A_{1,n}\}$.

[12] Die disjunktive Normalform ist dieselbe wie die konjunktive, nur & und $\vee$ vertauscht.

$\{A_{1,1}, \ldots, A_{1,n}\}$ ist eine **Klausel**. Die leere Menge ist immer falsch. Statt mit Formeln kann man mit endlichen Mengen von Klauseln arbeiten. Allgemeingültige Klauseln sind von der Form $\{P, \neg P\}$ und falsche von der Form $\{\}$. Ein Atom und seine Negation dürfen jetzt in einer Klausel vorkommen.

Eine **Hornklausel** ist eine Klausel mit höchstens einem positiven Literal (s. Kapitel 2). Verbindet man Hornklauseln durch Konjunktion oder negiert man sie, so kann man das Ergebnis wieder als Hornklausel schreiben. Die Disjunktion von Hornformeln ist aber nicht äquivalent zu einer Menge von Hornformeln. Wie soll $P \vee Q$ ausgedrückt werden? Hornformeln sind also keine Normalform, in die alle Aussagen übertragen werden können. Sie stellen ein Einschränkung dar, die es erleichtert, über die Erfüllbarkeit zu entscheiden.

3.2.5 Korrektheit, Vollständigkeit, Widerlegung

Wenn wir eine Behauptung nicht durch reine Anschauung begründen wollen und wenn wir die Behauptung allgemeiner Überprüfung zugänglich machen wollen, müssen wir festlegen, was wir als Begründungen zulassen wollen. Siegfried Kanngießer (1984) zitiert aus Leonhard Eulers "Briefen an eine deutsche Prinzessin über verschiedene Gegenstände der Physik und Philosophie" [13] über die guten Gründe, etwas für wahr zu halten:

1) Zeugnis der Sinne,

2) richtiger Schluß, reguläre Syllogismen,

3) Bericht von glaubwürdigen Personen.

In diesem Abschnitt geht es um den richtigen Schluß, die Deduktion. Wir haben bereits in Wahrheitstafeln gegründete Verknüpfungen und die logische Folgerung gesehen. Diese wurden im Ball-Beispiel dazu verwendet, eine Frage zu beantworten oder die Belegung aller Aussagensymbole zu finden, die alle Formeln wahr macht. Die Verwendung soll nun in Form von Regeln festgelegt werden. Die Regeln sollen Formeln nur unter Betrachtung ihrer Form in andere überführen, bis man das gewünschte Ergebnis hat. Die Regeln sollen so sein, daß genau und nur solche Ergebnisse herauskommen, wie man sie durch die logische Folgerung erhielte. Die Regeln sollen also die semantische Folgerung syntaktisch rekonstruieren. Die syntaktische Rekonstruktion der Folgerung heißt **Ableitung**. $X \vdash B$ heißt, daß B aus der Formelmenge X abgeleitet wird. Man muß dazu angeben, welche Regeln die Ableitung realisieren: $X \vdash B$ mit $\Re$. Formeln, Regeln und Axiome ergeben zusammen ein **Kalkül**. Axiome sollen möglichst wenige Formeln sein. In dem Ball-Beispiel war es nicht nötig, Gwahr als Axiom zu nehmen. Wegen der Axiome wird ein Kalkül auch axiomatische Theorie genannt.

Es gibt verschiedene Logikkalküle. Hier führe ich nur zwei Regeln an: die positive (mit W) und die negative (mit F) Schnittregel. Sie sehen so aus:

$$\Re: \qquad \frac{W \rightarrow P \quad P \wedge C \rightarrow D}{C \rightarrow D} \qquad \frac{A \rightarrow P \quad P \wedge C \rightarrow F}{A \wedge C \rightarrow F}$$

[13] Die Prinzessin war Sophie Friederike Charlotte Leopoldine Louise (Brandenburg-Schwedt). Sie erhielt Privatunterricht von Euler, der dann schriftlich fortgeführt wurde. Nur auf diese Weise konnte sie an den philosophischen Diskussionen über Leibniz' Monadenlehre und anderes teilhaben.

Wenn eine Regel aus einer Formelmenge X die Formel B ableitet und aus X folgt B, so ist die Regel **korrekt**. X |- B ist also korrekt, wenn X |= B. Eine Menge von Regeln ist korrekt, wenn jede ihrer Regeln korrekt ist.

Widerlegung: eine Formelmenge X wird widerlegt, indem F aus ihr abgeleitet wird. Eine Formel A wird bezüglich einer Formelmenge X widerlegt, indem F aus X,A abgeleitet wird.

Es reicht nicht aus, daß eine Menge von Regeln korrekt ableitet. Es könnte ja immer noch sein, daß einige logische Folgerungen nicht abgeleitet werden! Adäquat ist eine Menge von Regeln erst, wenn sie korrekt und vollständig ist.

Vollständig für das Ableiten ist eine Menge von Regeln, wenn gilt: immer wenn X |= B, dann auch X |- B. Vollständig für das Widerlegen ist eine Menge von Regeln, wenn gilt: immer wenn X |= F, dann auch X |- F.

$\Re$ ist nicht vollständig für das Ableiten, weil man aus der leeren Menge nichts ableiten kann - ergo fehlen die Tautologien -, aber vollständig für das Widerlegen von Hornformeln: wenn X |= F, dann X |- F. Dies kann durch einen Beweis mit Kontraposition gezeigt werden. Wir zeigen also zuerst, daß, wenn X nicht F ableitet, F auch nicht aus X folgt. Wir nehmen an, daß X nicht F ableitet. Wenn X nicht F ableitet, soll es erfüllbar sein. Also muß es eine Belegung β geben, die X wahr macht. Die konstruieren wir. Die erste Idee dazu ist: β(P) = W gdw. P $\in$ X. P soll eine Formel aus X sein. Q $\rightarrow$ P enthält P aber nur als Teilformel. Die Konstruktion gibt also noch nicht alles wieder und muß verbessert werden. Dazu bildet man ein X^* aus X vereinigt mit {P; P Atom, X |- P}. β(P) = W gdw. P $\in$ X^*. Jetzt sind alle Formeln und alles Ableitbare wahr. X ist erfüllbar. Die Semantik stimmt mit $\Re$ bezüglich der Widerlegung überein. Der genaue Beweis verwendet nur die positive Schnittregel und steht in Siefkes (1990: 61).

3.2.6 Literatur

*Bläsius, K.H., Bürckert, H.-J. (1987): Deduktionssysteme - Automatisierung logischen Denkens, München: Oldenbourg

Kanngießer, Siegfried (1984): Simulationskonzepte des Wissens und der Grammatik, in: Rollinger (ed): Probleme des (Text-)Verstehens, Tübingen: Niemeyer, S. 24-44.

Schöning, Uwe (1995): Logik für Informatiker, 4.Aufl. Heidelberg: Spektrum Akademischer Verlag.

*Siefkes, Dirk (1990): Formalisieren und Beweisen - Logik für Informatiker, Braunschweig: Vieweg

3.3 Termsubsumtionssysteme und ihre Vorläufer

Im letzten Kapitel haben wir gesehen, wie man Logik zum Problemlösen benutzen kann. In diesem Kapitel will ich zeigen, wie in der KI Logik benutzt wird, um einen Formalismus zu fundieren. Der Beschreibungsansatz der KI beschreibt ein Phänomen in einem Formalismus, der seinerseits Eigenschaften hat. Logik wird oft deshalb als Beschreibungssprache gewählt, weil wir einige ihrer Eigenschaften kennen und auf Jahrhunderte der Forschung zugreifen können. [14] Dabei ist aber die Logik für den Zweck der Problembeschreibung und für die Interaktion mit dem Benutzer eines Systems nicht so gut geeignet. Man kann dann einen anderen Formalismus konstruieren, den man wiederum in Logik beschreibt. Damit hat der Formalismus nicht nur eine Interpretation in Form eines Programms, das Ausdrücke des Formalismus verarbeitet, sondern auch eine Interpretation, die auf logische Formeln abbildet. Diese logische Fundierung macht das Verständnis eines Systems unabhängig vom Kennen des Programmes. Damit ermöglicht es eine bessere Kommunikation unter den WissenschaftlerInnen. Auch können Eigenschaften des Programms eingeteilt werden in unerwünschte (von der Logik nicht vorhergesagte) und erwünschte, was den SystementwicklerInnen hilft. Natürlich hat dies auch einen Nachteil: die Prädikatenlogik reicht nicht aus, um alles auszudrücken, was wir ausdrücken möchten! Weiterentwicklungen der Logik sind dann nötig. Bis dahin kann man in begründeten Fällen etwas operationalisieren, das noch nicht logisch fundiert ist.

Im folgenden werde ich zunächst den Hintergrund der Entwicklung von KL-ONE angeben: semantische Netze und Frames. KL-One-artige Formalismen heißen Termsubsumtions-Formalismen. Diese werde ich mithilfe der Logik beschreiben. Das Ganze ist ein Lehrstück der KI und zeigt, wie aus einer intuitiven Anschauung schließlich ein logisch fundierter Formalismus wird, der für den Benutzer aber immer noch die Anschaulichkeit besitzt. Das Lehrstück ist genau dokumentiert in Brachman & Levesque´s Sammelband (1985).

3.3.1 Der Hintergrund - semantische Netze

Semantische Netze wurden psychologisch motiviert eingeführt. Die Ausgangsbeobachtung war, daß Menschen bei Versprechern nicht beliebige Wörter verwechseln. Dies wurde nicht mehr als Freudsche Verdrängung gesehen, sondern informationstheoretisch aufgefaßt: es werden solche Wörter verwechselt, deren semantische Repräsentation sich nur in wenigen bits unterscheidet. Wie soll also so eine semantische Repräsentation aussehen? Man nehme Begriffe (die semantische Repräsentation von Wörtern) als Knoten und Beziehungen zwischen ihnen als Kanten. Der Zugriff auf einen Begriff ist ein Pfad durch einen solchen Begriffsgraphen. Bei nur einer Verzweigung sich zu irren, führt zu einem Versprecher. Das semantische Netz gibt also eine Speicherung von Begriffen an, die die Ähnlichkeit von Begriffen durch die Kürze eines Pfades von einem Begriff zum anderen ausdrückt.[15] Durch die Zusammenhänge der Knoten ergibt sich ihre Bedeutung. Die Verarbeitung geschah zunächst mit der Methode der *spreading activation*: von zwei Knoten ausgehend werden alle von ihnen abgehenden Zeiger aktiviert. Diese aktivieren dann die Knoten, auf die sie zeigen. Von diesen Knoten

14 Eine andere Form der Beschreibung wählt als formale Basis, über die bereits viel bekannt ist: neuronale Netze werden mit Differentialgleichungen beschrieben.

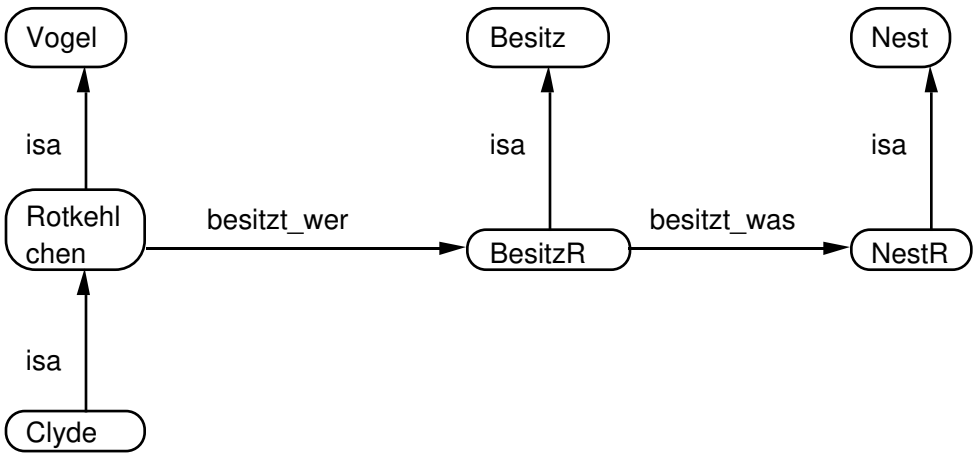
15 Die Doktorarbeit von Ross Quillian 1966 hieß "Semantic Memory". Sie führte semantische Netze ein. Die Kurzfassung kam 1967 in *Behavioral Science* heraus und ist im Sammelband von Brachman und Levesque nachgedruckt.

ausgehende Zeiger werden wiederum aktiviert. Dies wird solange gemacht, bis man eine Verbindung zwischen den beiden Ausgangsknoten gefunden hat. Diese Verbindung ist dann ein Knoten, der von beiden Ausgangsknoten aus aktiviert wurde. Die jeweiligen Pfade werden ausgegeben und sind ein Vergleich zwischen zwei Begriffen. Allerdings ist diese ungerichtete Aktivierung für praktische Anwendungen nicht geeignet und erlaubt nicht, bestimmte Schlußfolgerungen auszudrücken.

In der Folge wurden dann semantische Netze verbessert:

- die Knoten werden in einer Hierarchie angeordnet, so daß eine besondere Kante mit dem Namen `isa` oder `ako` (für "a kind of") vom Unter- zum Oberbegriff zeigt;
- entlang der `isa`-Kanten werden alle Eigenschaften (alle anderen Kanten) des Oberbegriffs an seine Unterbegriffe vererbt;
- Anfragen an ein semantisches Netz werden mithilfe des partiellen Abgleichs beantwortet.

Eine Anfrage an ein Netz wird als Teilnetz konstruiert, das mit dem Netz abgeglichen wird. Dabei erhalten alle Variablen die Werte, die sie zu einem erfolgreichen Abgleich brauchen. Nehmen wir z.B. das folgende semantische Netz:



An dieses Netz können jetzt etwa folgende Anfragen gestellt werden:

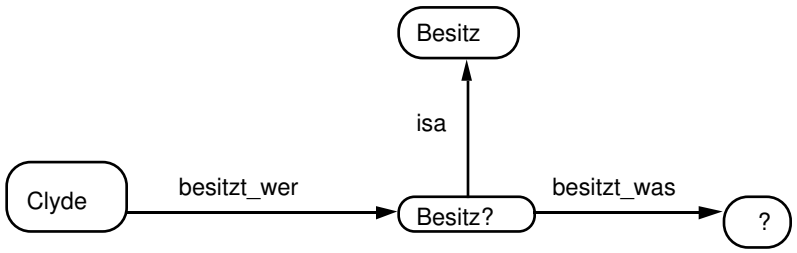
1) Was besitzt Clyde? etwa geschrieben:
besitzt_wer (clyde , besitzt_was (X, Y))

2) Besitzen alle Vögel ein Nest? etwa geschrieben: ∀ X|
isa (X, vogel) & besitzt_wer (X, besitzt_was (Y, Z)) & isa(Z, nest)

3) Gibt es einen Vogel, der ein Nest besitzt? etwa geschrieben: ∃ X|
isa (X, vogel) & besitzt_wer (X, besitzt_was (Y, Z)) & isa(Z, nest)

Zu jeder Anfrage muß jetzt für den Abgleich ein partielles Netz konstruiert werden. Das sind dann etwa die folgenden:

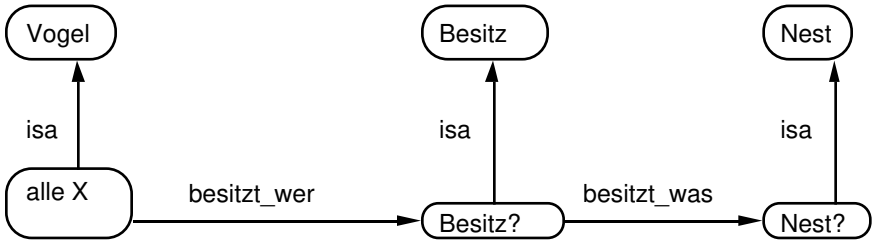
ad 1)



Zur Beantwortung muß der Abgleich mit dem Netz die Vererbung berücksichtigen. Dabei wird dann `X` an `besitzR` gebunden und `Y` an `nestR`. Die Antwort ist dann etwa: Clyde besitzt ein NestR oder `besitzt_wer (clyde, besitzt_was (besitzR, nestR))`. Der Abgleich berücksichtigt also die Hierarchie und die Vererbung.

Die Berücksichtigung von Schlußfolgerungen beim Abgleich wird noch deutlicher bei der Anfrage 2).

ad 2)



Hier muß nicht nur die `isa`-Hierarchie für den Abgleich verwendet werden, um dann `X` einmal an `Rotkehlchen` und dann `Y` an `besitzR` und `Z` an `nestR` zu binden, sondern dies muß für alle Unterbegriffe von `Vogel` getan werden. Der Abgleichsprozeß muß also ein Verfahren enthalten, das

```
∀ X | isa (X,vogel)
```

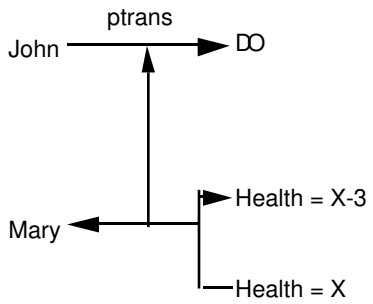
behandelt. In diesem Falle wäre die Antwort: ja.

Für die dritte Anfrage wird ein ähnliches Teilnetz konstruiert, nur daß die Variable X diesmal nicht all- sondern existenzquantifiziert ist. Die Antwort ist dann: ja, Rotkehlchen. Der Abgleichsprozeß muß die verschiedenen Quantoren von Variablen behandeln und Schlußfolgerungen ziehen können, weil das semantische Netz mit seiner speziellen Kante `isa` Regeln ausdrückt. Dadurch ist nicht alles direkt ablesbar, was das semantische Netz ausdrückt, sondern muß durch Schlußfolgerung explizit gemacht werden. Der Abgleich kann also ein Theorembeweiser sein.

An einer Darstellung von Begriffen und ihren Zusammenhängen sind natürlich auch die Linguisten interessiert. Wenn man die Wortsemantik im Zusammenhang ausdrücken kann und vielleicht auch eine Satzsemantik daraus konstruieren kann, kann man endlich natürlichsprachlichen Sätzen eine operationale Repräsentation zuordnen. Man begann also, die Semantik von Wörtern als semantisches Netz darzustellen, wobei die Knoten Begriffe und die Kanten Beziehungen zwischen Begriffen darstellten.

Der Gedanke einer "Normalform", eines festen Repertoires von Kantennamen, liegt dann nahe. Denn wenn man einfach an die Kante das natürlichsprachliche Wort schreibt, hat man nicht viel gewonnen. Es wurde viel über semantische Primitive geschrieben. Das sollten diejenigen Kanten sein, mit denen man alle Beziehungen zwischen Begriffen konstruieren kann. Das erste Primitiv war `isa`. Schank (1973) führte mit seinen *conceptual dependency networks* semantische Primitive für Handlungen ein, aus denen sich Beschreibungen aller

Handlungen zusammensetzen lassen sollten. So z.B. `mtrans` für "mental transfer" als geistiger Austausch und `ptrans` für "physical transfer" als Austausch von Dingen (Kaufen, Verkaufen, Schenken, Geben, Nehmen, ...) oder direkte physische Einwirkung auf etwas oder jemanden. Ein typisches Beispiel von Schank war "John hurts Mary." das er dann folgendermaßen repräsentierte:



Dabei steht der dicke Pfeil für `ptrans`, die Gesundheit wird auf einer 10-Punkte-Skala angegeben. Den Pfeilzielen können außerdem Rollennamen mitgegeben werden wie etwa `Akteur` (bei `John`), `Leidtragender/Nutznießer` (bei `Mary`), `Gegenstand` (bei `Health`), `Handlung` (bei `DO`). Bestimmte Kasus oder Präpositionen im Satz werden dann diesen Rollen zugeordnet.

Das Problem bei diesen Ansätzen, semantische Netze und *conceptual dependency networks*, ist, daß sehr unterschiedliche Dinge als Kanten auftreten.[16] So wurde etwa nicht zwischen Ober-Unterbegriffsrelation und Begriff-Instanzrelation getrennt: `isa` verband sowohl das *Rotkehlchen* mit dem *Vogel* wie auch das bestimmte Rotkehlchen *Clyde* mit *Rotkehlchen* . Auch der Unterschied zwischen `ptrans` und `mtrans` und `isa` war zunächst nur an diesen Namen zu erkennen, für die es dann jeweils eine eigene Prozedur geben mußte, die die entsprechende Kante verarbeitet. Diese Prozeduren waren nicht einfach miteinander vergleichbar. Deshalb kritisierte Hayes (1977) dieses Vorgehen als "pretend it's english" und Drew McDermott (1978) schlug den "**Gensym-Test**" vor: damit man nicht durch eine natürlichsprachliche Bezeichnung irregeleitet wird und mehr vermutet als durch die Prozedur realisiert wird, ersetze man alle Bezeichner durch automatisch generierte Symbole.

Ebenso unklar wie die Kanten waren die Knoten.[17] Sollen generische Begriffe ausgedrückt werden ("das Rotkehlchen als solches") oder alle Mitglieder des Begriffs durch einen Knoten repräsentiert werden (extensionale Begriffsrepräsentation)? Ist ein Unterbegriff dann eine echte Teilmenge des Oberbegriffs? Solche Festlegungen konnten obendrein nur schwer diskutiert werden, weil die Autoren verschiedener Repräsentationssysteme über ihre Arbeiten in wilder Mischung von programmiersprachlichen, anwendungsspezifischen, logischen und kognitiven Ausdrücken berichteten.

3.3.2 Frames

Die Idee bei einem **Frame** war, alles Wissen, das zu einem bestimmten Objekt oder einer bestimmten Situation der Welt gehört, zusammenzufassen. So sollte ein Begriff wie Kindergeburtstag nicht einfach als Unterbegriff von Geburtstagsfeier defi-

---

16 Der Aufsatz von William Woods "What's in a link?" erschien 1975 und ist auch im angeführten Sammelband nachgedruckt.

17 "What's in a Concept?" fragte Ron Brachman und schrieb über den epistemologischen Status von semantischen Netzen (ebenfalls nachgedruckt im Sammelband).

niert werden, sondern auch die typischen Ereignisse wie Topfschlagen oder Reise nach Jerusalem, die typische Dekoration wie Luftballons, die typische Kleidung und das übliche Essen (Süßigkeiten, Kuchen mit Kerzen) sowie die Geschenke dargestellt werden. Dabei sind diese nicht wirklich zwingend. Ein Kind kann in abgerissener Jeans und ohne Geschenk auf einen Geburtstag gehen, auf dem es Kartoffelsalat mit Würstchen gibt und Videos geguckt werden. Dies Ereignis wird aber als abweichend wahrgenommen. Egal, ob definierende Eigenschaft oder typisches Ereignis, das Zusammengehörige sollte auch zusammengefaßt repräsentiert werden. Ein Beispiel für den Kindergeburtstag (nach Minsky 1981):

```
birthday_party:

dress:            sunday best

present:    must please host; must be bought and gift-wrapped.


child's birthday_party:

isa:        birthday_party

games:            hide and seek, pin tail on donkey

decor:            balloons, favors, crepe_paper

party_meal:       cake, ice_cream, soda, hot_dogs

cake:       candles, blow_out, wish, sing birthday song

ice_cream:  standard three_flavor


Pauls's birthday_party:

instance_of:      child's birthday_party

dress:            Paul's blue suit

present:    kite_315

ice_cream:  vanilla_700
```

Hier ist also die Unterbegriffsrelation `isa` von der Beziehung zwischen einem Begriff und seiner Instanz (`instance_of`) getrennt. Die Instanz soll nur solche Eigenschaften haben, die Spezialisierungen der Eigenschaften des Begriffs darstellen. Dabei kann zwischen definierenden und attributiven (auch: kontingenten) Eigenschaften unterschieden werden. Die attributiven Eigenschaften müssen dann nicht unbedingt Spezialisierungen der Eigenschaften des Begriffs sein. Minskys Beispiel hat überhaupt nur attributive Eigenschaften, denn die Definition des Geburtstags liegt ja in dem Geburtsdatum eines Menschen.

Ein Frame hat bestimmte Eigenschaften, die für ihn relevant sind. Sie werden *slots* genannt. In einem *slot* steht ein Verweis auf einen anderen Frame (nicht-terminaler *slot*) oder direkt eine Zeichenkette (terminaler *slot*). Für die Einträge in einen *slot* gibt es **Einschränkungen**, die bei nicht-terminalen *slots* dadurch gegeben sind, daß der Eintrag eine Instanz eines Frames sein muß; bei terminalen *slots* ist die Einschränkung einfach durch eine Bereichsangabe gegeben (z.B. `integer [10, 50]` für den Wert des Geburtstagsgeschenkes). Bei attributiven *slots*

schränkt nur der Wertebereich die möglichen Einträge ein, auch wenn es sich um einen nicht-terminalen *slot* handelt. Man kann den Verweis auf den entsprechenden Frame als *slot*-Füller auch als Voreinstellung (default) auffassen, die dann durch einen Wert überschrieben werden kann.

Wie bei den Kanten gab es auch bei den *slots* die Frage nach einem festen Repertoire von *slots*, also nach semantischen Primitiven. Und auch hier war wieder die Frage nicht zu beantworten, was denn eigentlich `dress`, `present` oder `ice_cream` bedeuten, wenn wir nicht vorgeben, daß das System englisch versteht.

An einen *slot* kann eine spezielle Verarbeitungsprozedur "angeheftet" werden (procedural attachment oder demon). Diese Prozeduren werden von Anfragen an oder von neuen Eintragen in ein System von Frames ausgelöst. Sie heißen dann *if-needed* respektive *if-added* Prozeduren. Die *if-needed* Prozeduren berechnen der Wert des slots bei einer Anfrage für das spezielle, angefragte Objekt. Man spricht auch von *question-time inferences*. Zum Beispiel kann man den Umfang eines Kreises aus seinem Radius berechnen.

```
Kreis:

Radius:          real

Umfang:     if_needed: 2 π Radius
```

Es ist also eine Rückwärtsverkettung, mit der man den Wert eines *slots* ableitet.

Die *if-added* Prozeduren berechnen den speziellen *slot*-Wert für ein neu eingetragenes Objekt. Man spricht auch von *read-time inferences*. Zum Beispiel kann man für einen speziellen Kreis beim Eintragen des Wertes für den *slot* `Radius` sofort den Umfang berechnen und das Ergebnis bei `Umfang` eintragen.

```
Kreis:

Radius:          if_added: Umfang= 2 π Radius

Umfang:     real
```

Es ist also eine Vorwärtsverkettung, mit der man den Wert eines *slots* ableitet, der sich aus einem anderen Wert ergibt.

Die Prozeduren durchbrechen den expliziten Charakter der Repräsentation. Sie sind selbst nicht mehr durch Einträge veränderbar, sondern nur mithilfe eines Texteditors, und liegen nicht mehr so offen zutage wie die Wertrestriktionen durch Verweis auf einen anderen Frame oder durch eine Bereichsangabe.

Die Verarbeitung, Anfragen und Einträge, geschieht mithilfe von Zugriffsprozeduren auf Frames, ihre *slots* und deren Werte. Im einfachsten Falle werden die Namen von Frames und *slots* sowie die Werte terminaler *slots* abgeglichen. Im allgemeineren Fall kann eine Struktur in der Anfrage mit Namen im Frame-System abgeglichen werden, wobei die Vererbungshierarchie und die *if_needed*-Prozeduren ausgewertet werden. So kann gefragt werden, bei welchen Gelegenheiten Sonntags-Kleidung getragen wird. Oder auch, ob für Kindergeburtstage Geschenke gekauft werden. Oder auch, welchen Umfang ein Kreis mit bestimmtem Radius hat.

```
Frame?, dress: sunday best

child's_birthday_party, present: must be bought.
```

```
kreis_13, umfang: ?
```

Man kann einen Frame auch logisch beschreiben und dann entsprechend auch Anfragen und Einträge als Beweis auffassen. Für den Kindergeburtstag ergibt sich:

```
∀ X | frame(child's_birthday_party, X) --> frame(birthday_party, X)

∀ X | frame(child's_birthday_party, X) --> ∃ Y1 | dress(X,Y1) &
                                                  sunday_best(Y1)

∀ X | frame(child's_birthday_party, X) --> ∃ Y2 | games(X,Y2) &
                        (hide_and_seek(Y2) v pin_tail_on_donkey(Y2))

...
```

Entsprechend können dann die Anfragen formuliert werden:

```
∃ Z | frame(Z,X), dress(X,Y1), sunday_best(Y1)

∀ X | frame(child's_birthday_party, X), present(X,must_be_bought)

frame(kreis_13, X), radius(X,12), umfang(X,Value)
```

Bei dem speziellen `kreis_13` ist es unsinnig, von allen Instanzen des Frames zu sprechen, weil er bereits eine Instanz ist. Daher ist bei der letzten Anfrage kein Quantor angegeben.

Die logische Beschreibung hat `child's_birthday_party` zu einem Argument von `frame` gemacht. Damit können Anfragen nach Frames ganz allgemein operationalisiert werden. Es bleiben aber noch `dress`, `sunday_best` und `present` als spezielle Prädikate, deren Status unklar ist.

Bestimmte Prozeduren werten Anfragen und Einträge aus.

Die Prozeduraufrufe (in Prolog: Klauselköpfe) wären für Anfragen etwa

```
ask_frames(Formelliste),

ask_truth (Formelliste),

ask_value(Formelliste)
```

und die konkreten Aufrufe für die Anfragen von oben wären:

```
ask_frames([frame(Name, X), dress(X,Y1), sunday_best(Y1)])

ask_truth([frame(child's_birthday_party, X), present(X, must_be_bought)])

ask_value([frame(kreis_13, X), radius(X, 12), umfang(X,Value)]).
```

Für Einträge gäbe es beispielsweise die Prozeduren `tell_frame` für den Eintrag eines kompletten Frames, `tell_slot` für den Eintrag eines zusätzlichen *slot* , `tell_value_restriction` für den Eintrag einer Einschränkung, `tell_value` für den Eintrag eines neuen Wertes für einen *slot*.

Die unterschiedlichen Frame-Systeme unterscheiden sich in ihren Festlegungen. Entsprechend sind die Zugriffsprozeduren, d.h. die Verarbeitung unterschiedlich. Ein Problem der Frames war ferner, daß mit den angehefteten Prozeduren *if_needed* und *if_added* der explizite Charakter der Repräsentation verlassen

wurde und die Ebene der Programmierung direkt in den Formalismus einbezogen wurde. Wie bei den semantischen Netzen waren die Festlegungen zunächst nicht genau definiert: was ist ein Frame, was ist ein *slot*, wie werden die möglichen Einträge eingeschränkt, wie verhalten sich definierende und attributive Eigenschaften?

3.3.3 Zur Beschreibung von semantischen Netzen und Frames

Im Laufe der Diskussionen und Weiterentwicklungen von semantischen Netzen und Frames wurde deutlich, daß sie eigentlich nur notationelle Varianten voneinander sind. Man erkennt das, wenn man für beide die Bedeutung ihrer Konstrukte (*slot*, Kante, Frame, Knoten) logisch formuliert. Dann sind die Festlegungen (etwa Wertebeschränkungen bei *slot*-Werten, Wertebeschränkungen bei Ursprungs- und Zielknoten einer Kante) implementationsunabhängig beschrieben und diskutierbar. Das eben fehlte ja zunächst!

Es wird dann auch klar, was den unklaren Status der Einheiten des Repräsentrationsformalismus' (Kanten, *slots*) ausmachte: durcheinander gingen die verschiedenen Ebenen, auf denen man über dieselbe Sache sprechen kann:

die **Implementationsebene**, auf der eine Kante ein Name und ein(LISP)Zeiger, ein Knoten eine Liste und das Abgleichsprogramm eine Sammlung von Prozeduren für verschiedene Kantentypen ist,

die **logische Ebene**, auf der die Kanten Prädikate (oder Operatoren?) und die Knoten Terme (oder Mengen von Konstanten?) sein können,

die **begriffliche Ebene**, auf der Wort- oder Satzbedeutungen gemeint sind, wobei Kanten verschiedene Rollenbeziehungen und Knoten Begriffe darstellen,

die **sprachliche Ebene**, auf der die Kanten zum Beispiel Verben und die Knoten Substantive darstellen, Kantentypen vielleicht (Tiefen-)kasus zugeordnet sind.

Diese Ebenen können aufeinander aufbauen, wenn sie jeweils für sich wohl definiert sind. Die Vermittlung zwischen logischer Ebene und begrifflicher Ebene war zunächst nur durch die semantischen Primitive bzw. ihre Prozeduren definiert. Um dies klarer zu fassen, führte Brachman 1979 die **epistemische Ebene** ein (der Artikel ist ebenfalls im Sammelband von Brachman und Levesque nachgedruckt). Diese Ebene entspricht der Wissensebene von Newell. Auf dieser Ebene wird das begrenzte Repertoire von epistemischen Primitiven definiert. Mit den epistemischen Primitiven lassen sich dann Begriffe und ihre Beziehungen definieren. Sie beziehen sich auf das Definieren, nicht auf das Definierte. Tatsächlich war `isa` eigentlich ein epistemisches Primitiv: gibt man diesem eine klare Bedeutung, nach der es von verschiedenen Abgleichsverfahren, realisiert in verschiedenen Programmiersprachen, immer gleich interpretiert wird, so kann es sinnvoll und vergleichbar auf die Einheiten der begrifflichen Ebene angewandt werden. Da das Definieren selbst weniger vielfältig ist als all das, was man alles definieren mag, kann man hoffen, mit einem überschaubaren Kanon von epistemischen Primitiven auszukommen. Zum Beispiel kann man mit `isa` und `role` auskommen, wobei `isa` die Oberbegriff-Unterbegriffsrelation und `role` die definierenden Eigenschaften darstellt. Eine Begriffsdefinition ist dann immer eine Hierarchie von Begriffen, wobei die gemeinsamen Eigenschaften aller Unterbegriffe beim Oberbegriff angegeben werden und die unterscheidenden Eigenschaften bei den jeweiligen Unterbegriffen. Dies ist das Descartesche Modell der Definition: erst gibt man den allgemeinen Begriff, dann die unterscheidenden Eigenschaften.

Ein Beispiel zeigt die Ebenen und den Unterschied der epistemischen Ebene zur begrifflichen und logischen. Wir übernehmen die Aussagen aus dem Rotkehlchen-Nester-Beispiel. Von derselben Sache werden jeweils unterschiedliche Aspekte betrachtet:

Auf der **epistemischen Ebene** sprechen wir von Oberbegriff-Unterbegriffsrelation `isa`, von definierenden Eigenschaften und Vererbung von Eigenschaften. Zum Beispiel: Vogel ist ein Oberbegriff von Rotkehlchen, etwas zu besitzen ist eine Eigenschaft von Rotkehlchen. Wir legen fest, was mit den epistemischen Primitiven gemeint ist. Zum Beispiel: Ein `Begriff` ist die Entscheidung, welche Instanzen zu seiner Extension gehören. Die `isa`-Relation zwischen Begriffen drückt aus: Ein Oberbegriff deckt mindestens alle Instanzen der Unterbegriffe ab. `role` drückt definierende Eigenschaften aus, indem es zwei Begriffe verbindet. Dabei gilt: alle Instanzen des Begriffs müssen eine Eigenschaft haben, die eine Instanz des anderen Begriffs ist.

Auf der **begrifflichen Ebene** geht es um die Begriffe des Sachbereichs, die wir repräsentieren. In unserem Beispiel also um Rotkehlchen, Nester, Besitzverhältnis und Clyde. Zum Beispiel: `Vogel` ist ein Oberbegriff von Rotkehlchen, `besitzt_was` ist eine Eigenschaft von Rotkehlchen.

Auf der **logischen Ebene** sprechen wir von Prädikaten, Termen, Quantoren und Variablen. Wir können bei Prolog für die Implementierung oft die Ausdrücke der logischen Ebene direkt auf die Implementierungsebene übertragen - aber nicht immer! Zum Beispiel: `∀ X | isa(X, rotkehlchen) --> isa(X,vogel).` Wir beschreiben die Festlegungen der epistemischen Ebene logisch. Zum Beispiel fassen wir `role` als eine Funktion auf, die allen Instanzen eines Begriffs eine Instanz eines anderen Begriffs zuordnet.

Der Einfachheit halber können wir Prolog als Interpreter verwenden. Das ist dann die **Implementationsebene**. Auf dieser Ebene reden wir von cuts und Aufrufen und eingebauten Prädikaten. Wir verwalten files. Daß wir hier auch von Klauseln sprechen, liegt daran, daß Prolog zum Ziel hatte, die logische Ebene direkt zu operationalisieren. Wir können aber auch LISP oder C oder Pascal verwenden und sprechen dann von Listen und Funktionen. Wir operationalisieren die Festlegungen der epistemischen Ebene, indem wir für `isa` und `role` Klauseln schreiben, die Netzanfragen und -einträge entsprechend auswerten. Die Klauselköpfe sind zum Beispiel: `ask(isa(X,Y)), tell(role(Name, Range, Value)).`

Wir könnten etwa folgende Einträge in einer Prolog-Basis haben, die das Vogel-Nest-Netz ausdrücken:

```
isa(rotkehlchen, bird). isa(nestR, nest). isa(besitzR, besitz).

role(besitzt_wer, rotkehlchen, besitzR). role(besitzt_was,besitzR,nestR).
```

Die Instanz `clyde` wäre dann

```
instance(clyde, rotkehlchen). role(besitzt_wer, clyde, besitzC).
```

Man kann dann folgern

```
role(besitzt_was, besitzC, instance(X,nestR)).
```

Das Beispiel zeigt:

- Wichtig ist, daß wir nicht geschrieben haben `besitzt_wer(rotkehlchen, besitzR)`, weil wir sonst für jede Kante eine eigene Klausel zur Verarbeitung schreiben müßten. So brauchen wir nur jeweils eine für `isa` und `role`

zu schreiben. Die Verarbeitungsprozeduren beziehen sich jetzt auf epistemische Primitive.

- Wichtig sind nicht die jeweiligen Festlegungen, die in dem Beispiel angegeben sind, sondern daß man sie explizit und implementationsunabhängig treffen kann. Dabei verschwinden Entitäten der Implementationsebene aus den übergeordneten Ebenen. Der Bruch, der bei den *if_needed*- und *if_added*-Prozeduren auftrat, geschieht nicht mehr. Allerdings sind damit für Einzelfälle mögliche Tricks, die man in einem Programm unterbringen, aber nicht für alle Fälle in ihrer Bedeutung definieren kann, nicht mehr vorhanden. Die Transparenz kann zulasten einer punktuellen Systemleistung gehen.

- Alles, was wir in Logik nicht ausdrücken können, geht nicht. So haben wir keine attributiven Werte (Relationen zwischen Knoten) mehr. Erst eine andere als die Prädikatenlogik erster Stufe kann vielleicht auch Voreinstellungen (defaults) ausdrücken, so daß wir sie dann wieder in das System hineinnehmen können.

Wenn die begriffliche Ebene die Definition von Begriffen darstellt, kann sie nicht die Beziehung zwischen Begriff und Instanz darstellen. Die Aussagen über Objekte (Instanzen, Individuen) der Welt müssen getrennt werden von Aussagen über Klassen (Begriffe). Der Teil eines semantischen Netzes oder Frame-Systems, der Begriffe definiert, heißt bei Brachman terminologisch oder auch **T-Box**. Die Verarbeitung innerhalb der T-Box ordnet einen definierten Begriff in die Begriffsstruktur ein. Der Teil, der Individuen speichert, heißt assertional oder auch **A-Box**. Die Verarbeitung innerhalb der A-Box klassifiziert (bei Termsubsumtions-Formalismen wird das oft "realisiert" genannt) ein Grundbeispiel, eine Instanz. Ein System, das beide Teile und ihre Verbindung verwaltet, heißt hybrides System.

3.3.4 Termsubsumtions-Formalismen

Terminologische Wissensrepräsentationsformalismen (also: eine T-Box) gründen sich auf die Teilmengenbeziehungen ihrer Begriffe. Ein Oberbegriff subsumiert einen Unterbegriff. Alle Instanzen des Unterbegriffs sind auch Instanzen des Oberbegriffs. Unsere Termini sind **Begriffe** und **Rollen**. Die Subsumtion von Begriffen und Rollen ist der Kern des Repräsentationsformalismus. Dabei gibt es in vielen solchen Formalismen noch eine zusätzliche Konstruktion, die so dringend gebraucht wird, daß sie trotz ihrer Komplexität einbezogen wird: die **Anzahlrestriktion** oder Kardinalität. Eine Rolle bekommt dann neben dem Namen, dem Ursprungsbereich und dem Wertebereich noch eine Kardinalität als Angabe. So können wir ausdrücken, daß jedes Rotkehlchen genau ein Nest besitzt, jeder Mensch genau eine Nase, zwei Augen, zwei Arme, zehn Finger. Wir können auch Intervalle als Anzahlrestriktion angeben: ein Fahrrad hat mindestens eins und höchstens vier Räder.

Wir können jetzt die Syntax und Semantik genau definieren und erhalten damit ein wohldefiniertes System, dessen Eigenschaften wir untersuchen können.

3.3.4.1 Syntax eines Termsubsumtions-Formalismus'

Es werden Begriffe und Rollen jeweils nur einmal eingeführt. Ein Begriff wird als primitiver Begriff eingeführt, indem man nur den Oberbegriff dazu angibt, aber keine weitere Rolle. Die Schreibweise ist: `BegriffNeu ≤ Oberbegriff`. Ein Begriff wird als definierter Begriff eingeführt, indem man schreibt: `BegriffNeu .=. <Begriffsdefinition>`, wobei die Begriffsdefinition durch die Angabe des Oberbegriffs und der definierenden Rollen geschieht. Alle Rollen des Oberbegriffs

werden an den Unterbegriff vererbt. Man kann auch Rollen einführen, denn auch Rollen befinden sich jetzt in einer Hierarchie. Sie können als primitive Rollen eingeführt werden: `RolleNeu ≤ Oberrolle`, oder als definierte Rolle durch Angabe des Wertebereichs und der Anzahlrestriktion: `RolleNeu .=. Oberrolle`.

Man kann neue Begriffe und Rollen bilden. Begriffsbildende Operationen sind:

`and` für die Begriffskonjunktion. So kann aus den Begriffen `Frau` und `StudentIn` der Begriff `Studentin` gebildet werden.

`all` für die Werterestriktion. Das ist die Beschränkung des Wertebereichs, also des Begriffs, auf den eine Rolle zeigen darf. So kann die Rolle `besitzt_wer` nur mit etwas gefüllt werden, was ein Besitz ist.

`atleast` und `atmost` für die Anzahlrestriktion. So können wir ausdrücken, daß ein Rotkehlchen `atleast` 1, `atmost` 1 Besitz hat.

Wenn Begriffe vollständig definiert sind, ist an ihnen abzulesen, ob sie disjunkt sind oder nicht. Bei primitiven Begriffen können wir die Angabe als Restriktion hinzufügen, daß sie disjunkt sein sollen.

Entsprechend könnten wir auch Rollen zusammensetzen. Einige Formalismen tun das. Hier geht es aber darum, den Kern von Termsubsumtions-Formalismen darzustellen, nicht die vielfältigen Varianten.

Die Syntaxdefinition für einen Termsubsumtions-Formalismus gibt Nebel (1990: 48) in Backus-Naur-Form an:

```
          <terminology> ::=      { <term-introduction> | <restriction>}*

 <term-introduction> ::=      <concept-introduction> | <role-introduction>

       <concept-introduction> ::=   <atomic-concept> .=. <concept> |

                          <atomic-concept> ≤ <concept> |

                            <atomic-concept> ≤ ANYTHING

             <role-introduction> ::=      <atomic-role> .=. <role> |

                            <atomic-role> ≤ <role> |

                           <atomic-role> ≤ ANYRELATION

                       <concept> ::=    <atomic-concept> |

                                (and <concept>+) |

                             (all <role> <concept>) |

                           (atleast <number> <role>) |

                             (atmost <number> <role>)

                   <role> ::= <atomic-role> | (androle <role>+)

       <restriction> ::=      (disjoint <atomic-concept> <atomic-concept>)

                     <number>  ::=    <non-negative integer>
```

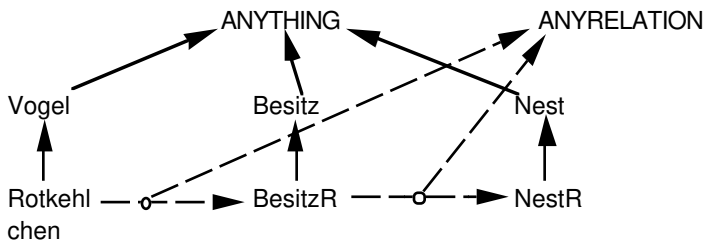
```
<atomic-role>  ::=     <identifier>

<atomic-concept>  ::=  <identifier>
```

Dabei sind ANYTHING und ANYRELATION der allgemeinste Begriff bzw. die allgemeinste Rolle. So kann also ein neuer primitiver Begriff als Unterbegriff von ANYTHING eingeführt werden. androle bildet den Schnitt mehrerer Rollen, womit jetzt auch Rollen definiert eingeführt werden.

Das Vogelbeispiel mit der Ergänzung, daß ein Rotkehlchen nur genau ein Nest besitzt, sieht in dieser Syntax so aus:

```
Vogel ≤ ANYTHING  Besitz ≤ ANYTHING Nest ≤ ANYTHING

NestR ≤ Nest

besitzt_wer ≤ ANYRELATION     besitzt_was ≤ ANYRELATION

BesitzR .=. (and Besitz (all besitzt_was NestR)

      (atleast 1 besitzt_was) (atmost 1 besitzt_was))

Rotkehlchen .=. (and Vogel (all besitzt_wer BesitzR))
```

Die isa-Relation ist nicht als Rolle notiert, sondern ergibt sich bei Primitiven durch ihre Einführung und wird bei Definierten (wie hier BesitzR) durch die Angabe des Oberbegriffs angeben (erste Angabe im and-Teil). Die Vererbung von Eigenschaften entlang der Begriffshierarchie ist damit weiterhin gegeben. Bei diesem Ausschnitt werden die Rollen nicht näher definiert, sie müssen aber "oberhalb" ihrer Verwendung eingeführt worden sein. Das besitzt_was zwischen BesitzR und NestR ist natürlich spezieller als das zwischen Besitz und Nest, weil die Begriffe jeweils Oberbegriffe sind. Die Aussage ist hier, daß ein Rotkehlchen ein Vogel ist, der etwas besitzt. So ist Rotkehlchen hier definiert. Ein anderer Name würde der Definition besser entsprechen, vielleicht Nestbrüter.



Da es innerhalb einer T-Box nur um die Definition von Begriffen geht, fehlt Clyde in dem Netz.

3.3.4.2 Semantik eines Termsubsumtions-Formalismus

Ganz wesentlich für den Fortschritt gegenüber ursprünglichen semantischen Netzen und Frames war die logische Interpretation. Wir geben also jedem verwendeten epistemischen Primitiv sowie allen Möglichkeiten, sie zusammenzusetzen, eine wohldefinierte Bedeutung. Dann können wir Eigenschaften des Formalismus untersuchen und erkennen, welche unterschiedlich aussehenden Formalismen eigentlich gleich sind, und welche nicht.

Geben wir nun also die formale Semantik für eine T-Box so an, wie sie von Bernhard Nebel (1990) für den Kern von Termsubsumtions-Formalismen ausgearbeitet wurde. Der Grundgedanke dieser Formalismen war ja, Begriffe zu subsumie-

ren, d.h. Mengen möglicher Instanzen in Teilmengenbeziehungen zu setzen. Die Interpretation von Begriffen ist also der Verweis auf ihre möglichen Instanzen. Da in der T-Box keine konkreten Instanzen angegeben sind, muß man auf notwendige Beziehungen aller Instanzen verweisen. Wenn etwa Vogel ein Oberbegriff von Rotkehlchen (oder Nestbrüter) ist, so interpretieren wir das als: ∀ x `rotkehlchen(X) --> vogel(X)`. Für die formale Notation nehmen wir eine Extensionsfunktion `ext` an, die Begriffe auf Mengen von Objekten (eben:Instanzen) abbildet, und Rollen auf Teilmengen des kartesischen Produktes der Objekte (also Objekt-Tupel)[18]. Die Funktion `ext` interpretiert uns also die Ausdrücke unserer Signatur, indem sie auf logische Strukturen abbildet. D sei die Menge von Objekten, c sei ein Begriff, r sei eine Rolle, a sei ein atomarer Begriff oder eine atomare Rolle, t irgendein Begriff oder irgendeine Rolle. Die Extensionsfunktion tut nun folgendes:

`ext`

`für alle a ≤ t sei ext(t) ⊇ ext(a)`

`für alle a .=. t sei ext(t) = ext(a)`

`ext( ANYTHING) = D`

`ext(ANYRELATION) = D × D`

`ext( (and c1 ... cn)) = ∩ ext(ci) wobei i von 1 bis n`

`ext( (all r c)) = {x ∈ D | ∀y : (x,y) ∈ ext(r) --> y ∈ ext (c)}`

`ext( (atleast n r)) = { x ∈ D | card ( {y ∈ D | (x,y) ∈ ext (r)} ) ≥ n }`

`ext( (atmost n r)) = {x ∈ D | card ( {y ∈ D | (x,y) ∈ ext (r)} ) ≤ n }`

`ext( (androle r1 ... rn)) = ∩ ext(ri) wobei i von 1 bis n`

`für alle (disjoint c1 c2) sei ext(c1) ∩ ext(c2) = {}`

Dabei liefert `card` die Kardinalität der Menge. Diese Interpretationsvorschriften sind fast schon in Ordnung. Sie lassen allerdings noch zyklische Definitionen zu. Das sind Definitionen von Begriffen, die den Begriff selbst in seiner Definition direkt oder indirekt verwenden. Tatsächlich geschieht es leicht, daß ein indirekter Zyklus vom Benutzer eingegeben wird: ein Nestbrüter hat einen Besitz, der ein Nest ist und ein Nest ist der Brutort eines Nestbrüters. Die meisten Termsubsumtions-Formalismen verbieten definitorische Zyklen[19].

Wir können jetzt die **Termsubsumtion** definieren, wobei wir auf die logische Struktur, die durch die Menge der Objekte D und die Extensionsfunktion `ext` gegeben ist, verweisen. Wir machen Aussagen über alle logischen Strukturen, weil wir ja notwendige Beziehungen zwischen allen möglichen Mengen von Instanzen angeben.

t subsumiert t' in einer T-Box T , geschrieben t » t', gdw.

für jede logische Struktur (D, ext) von T gilt ext(t) ⊇ ext(t').

---

18 "Objekte" wird hier wie in der Logik als Elemente des *universe of discourse* verstanden.

19 Nebel (90) gibt allerdings auch für eingeschränkte Zyklen eine Semantik an, so daß sie dann verarbeitet werden können; s. auch Baader (90).

Subsumtion ist eine transitive (d.h. wenn t » t' und t' » t'' dann t » t'') und reflexive (d.h. t » t) Beziehung, so daß man eine partielle Ordnung aller Begriffe und Rollen einer T-Box mithilfe der Äquivalenzrelation von Begriffen sowie Rollen hinbekommt. Die Äquivalenzrelation für eine T-Box T heißt, daß für zwei Begriffe oder zwei Rollen t und t' gilt:

t äquivalent t' gdw. in jeder logischen Struktur (D, ext) von T gilt ext(t) = ext(t').

Schließlich können wir noch die Inkohärenz eines Begriffs t in einer T-Box T feststellen:

t ist inkohärent in T, gdw. für jede logische Struktur gilt: ext(t) = {}

Inkohärente Begriffe sind also in sich selbst widersprüchlich und nicht nur in einer speziellen Struktur falsch.

`(and (atleast 1 r) (atmost 0 r))`

ist ganz sicher inkohärent. Wir können `NOTHING` als speziellsten Term in die T-Box einführen. `NOTHING` entspricht der leeren Menge. Mit den Schnittmengen (`and`) und `NOTHING` bildet die T-Box einen Halbverband, bei dem wir immer das Infimum bestimmen können.[20]

Für die Einordnung von Begriffen in einer T-Box können wir alle eingeschachtelten Definitionen glätten, indem wir für jeden definierten Begriff seine Definition einsetzen, wo immer er vorkommt. Das klappt, wenn wir keine Zyklen in den Definitionen haben. Die geglätteten Ausdrücke enthalten nur noch primitive Begriffe. Beim Glätten werden inkohärente Ausdrücke durch `NOTHING` ersetzt. Bei geglätteten Ausdrücken können wir dann feststellen, welcher Ausdruck von welchem subsumiert wird. Ein Algorithmus, der Begriffe in eine T-Box einordnet, heißt **Klassifikationsalgorithmus** (classifier). Schlußfolgerungen in der T-Box sind stets Klassifikationen. Da alle Begriffe gleich bei ihrem Eintrag in die T-Box klassifiziert (an die richtige Stelle im Begriffsnetz eingefügt) werden, kann eine Anfrage nach der Klassifikation eines Begriffes durch einfaches Traversieren schnell erfolgen. Ein Begriff wird in eine T-Box eingefügt durch die folgende Prozedur:

`classify(c)`

1) Die Begriffsdefinition wird in eine Form gebracht, die auf oberster Ebene einen `and`-Ausdruck hat, in den keine weiteren `and`-Ausdrücke eingebettet sind. Jeder verwendete Begriff in dem Ausdruck wird durch seine Definition ersetzt.

2) In der Begriffsbeschreibung werden inkonsistente Ausdrücke durch `NOTHING` ersetzt.

3) Wenn die Begriffsbeschreibung eine Werterestriktion ist (all r c), wird ein anonymer Begriff erzeugt, der gerade durch diese Wertebeschränkung definiert ist.

4) Die Definitionsbestandteile werden klassifiziert, d.h. `classify` wird rekursiv angewandt.

---

[20] Das heißt, für jede zwei Begriffe ist ihr größter gemeinsamer Unterbegriff angebbar: wenn sie keinen Schnitt haben, ist es die leere Menge, `NOTHING`.

5) Die T-Box wird traversiert, um den Platz zu finden, an den der Begriff gehört: unter alle direkten Oberbegriffe und über alle direkten Unterbegriffe. Wir suchen also die Begriffe super-c_i, für die gilt: `subsumes (super-`c_i`, c)`. Außerdem suchen wir die Begriffe sub-c_i, für die gilt: `subsumes(c, sub-`c_i`)`. Dabei verwenden wir die bereits aufgebaute Ordnung von Begriffen, müssen also nicht bei jedem Vergleich die Regeln von `subsumes` anwenden.

6) Der Begriff wird an dem in Schritt 5) bestimmten Platz eingefügt. Falls dort bereits ein Begriff eingeordnet ist, die beiden Begriffe also äquivalent sind, so fallen sie zu einem Begriff zusammen.

Die Traversierung (Schritt 5) kann gleichzeitig von den Blättern nach oben und von ANYTHING nach unten erfolgen. Der Kern des Klassifikationsalgorithmus' ist die Prozedur `subsumes`, die für zwei beliebige Begriffe t und u `wahr` oder `falsch` liefert, je nachdem ob t u unmittelbar subsumiert, oder nicht[21].

subsumes(t, u)

Wenn u = NOTHING dann liefere `wahr`.

Wenn t= (`and` t_1 ... t_n) oder t=(`androle` t_1 ... t_n), dann liefere `wahr`, wenn für alle i von 1 bis n `subsumes`(t_i, u) `wahr` liefert -- sonst liefere `falsch`.

Wenn t primitiv ist, dann

falls u primitiv ist, liefere `wahr`, wenn t=u -- sonst liefere `falsch`.

falls u = (`and` u_1 ... u_n) oder u=(`androle` t_1 ... t_n), liefere `wahr`, wenn es ein u_i gibt, so daß t = u_i , 1≤ i ≤n -- sonst liefere `falsch`.

sonst liefere `falsch`.

Wenn t = (`all` r_t c_t), dann

falls u = (`all` r_u c_u), liefere `wahr`, wenn sowohl `subsumes` (c_t c_u) als auch subsumes (r_u, r_t) `wahr` sind -- sonst liefere `falsch`.

falls u = (`atmost 0` r_u), liefere `wahr`, wenn `subsumes` (r_u, r_t) gilt -- sonst liefere `falsch`.

falls u = (`and` u_1 ... u_n), liefere `wahr`, wenn es ein u_i gibt, so daß subsumes (t, u_i) für 1≤ i ≤n gilt -- sonst liefere `falsch`.

sonst liefere `falsch`.

Wenn t = (`atleast` n_t r_t), dann

falls u = (`atleast` n_u r_u), liefere `wahr`, wenn sowohl subsumes (r_t, r_u) als auch $n_t \leq n_u$ gilt -- sonst liefere `falsch`.

falls u = (`and` u_1 ... u_n), liefere `wahr`, wenn es ein u_i gibt, so daß gilt subsumes (t, u_i), 1≤ i ≤n -- sonst liefere `falsch`.

---

21 Auch der Kern des *classifier* ist - leicht modifiziert - Nebel (90:76f) entnommen. Er basiert auf dem BACK-System, das an der TU Berlin entwickelt wurde (v.Luck et al. 87).

sonst liefere `falsch`.

Wenn t = (`atmost` n_t r_t), dann

falls u = (`atmost` n_u r_u), liefere `wahr`, wenn sowohl subsumes (r_u, r_t) als auch $n_u \leq n_t$ gilt -- sonst liefere `falsch`.

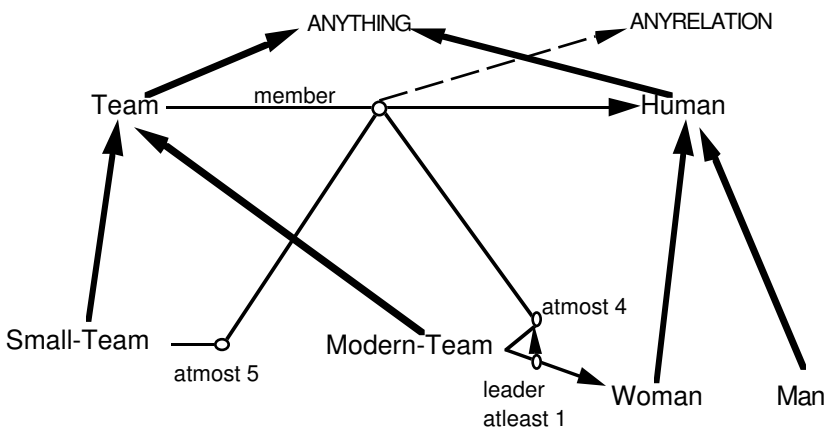
falls u = (`and` u_1 ... u_n), liefere `wahr`, wenn es ein u_i gibt, so daß gilt subsumes (t, u_i), $1 \leq i \leq n$ -- sonst liefere `falsch`.

sonst liefere `falsch`.

Dieser Algorithmus läßt sich in der folgenden Entscheidungstabelle übersichtlich zusammenfassen.

t / u	androle $t_1...t_n$ and $t_1...t_n$	primitiv	all r_t c_t	atleast n_t r_t	atmost n_t r_t
NOTHING	wahr	wahr	wahr	wahr	wahr
primitiv	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	t = u	falsch	falsch	falsch
all r_u c_u	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	falsch	subsumes (c_t,c_u)$\wedge$ subsumes (r_u,r_t)	falsch	falsch
atmost 0 r_u	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	falsch	subsumes (r_u,r_t)	falsch	falsch
and $u_1...u_n$ androle $u_1...u_n$	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	$\exists i, 1 \leq i \leq n$: t = u_i	$\exists i$, $1 \leq i \leq n$: subsumes (t,u_i)	$\exists i$, $1 \leq i \leq n$: subsumes (t,u_i)	$\exists i$, $1 \leq i \leq n$: subsumes (t_1,u)
atleast n_u r_u	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	falsch	falsch	subsumes(r_t,r_u) $\wedge$ $n_t \leq n_u$	falsch
atmost n_u r_u	$\forall_i$, $1 \leq i \leq n$: subsumes(t_i,u)	falsch	subsumes (r_u,r_t) bei n_u=0 sonst falsch	falsch	subsumes (r_u,r_t) $\wedge$ $n_u \leq n_t$

Das Beispiel, das sich durch das Buch von Nebel (1990) hindurchzieht, läßt sich folgendermaßen bildlich darstellen:



Es ist ein recht komplexes Beispiel, weil es eine Rollenspezialisierung enthält (`leader` ist spezieller als `member`) und weil es Anzahlbeschränkungen enthält (ein

kleines Team hat 5 Mitarbeiter, ein modernes Team hat vier Mitarbeiter und eine Leiterin).

Hinter der Auswertung von t = (all r_t c_t) steht die folgende Überlegung:

Wenn ext(c_t) $\supseteq$ ext(c_u) und ext(r_u) $\supseteq$ ext(r_t), dann muß gelten:

ext((all r_t c_t)) $\supseteq$ ext((all r_t c_u)) $\supseteq$ ext((all r_u c_u)).

Ein mit derselben Rolle definierter Begriff kann spezieller sein, wenn der Wertebereich spezieller ist:

Sei t = (all member Human) und u = (all member Woman), dann ist

ext(Human) $\supseteq$ ext(Woman) und ext (member) = ext (member)

woraus folgt:

ext((all member Human)) $\supseteq$ ext((all member Woman)) =

ext((all memberWoman))

Sogar ein mit einer allgemeineren Rolle definierter Begriff kann spezieller sein, wenn nur der Wertebereich spezieller ist:

Sei t = (all leader Human) und u = (all member Woman), dann ist

ext(Human) $\supseteq$ ext(Woman) und ext (member) $\supseteq$ ext (leader)

woraus folgt:

ext((all leader Human)) $\supseteq$ ext((all leader Woman)) $\supseteq$

ext((all member Woman))

Ein nur durch eine speziellere Rolle definierter Begriff kann nicht untergeordnet werden:

Sei t = (all member Human) und u = (all leader Woman), dann ist

ext(Human) $\supseteq$ ext(Woman) und $\neg$ (ext (leader) $\supseteq$ ext (member))

also subsumiert t nicht u! Denn es ist ja nichts über den Wertebereich von member bei u ausgesagt - der könnte größer sein als Human. Das Modern-Team ist deshalb ein Unterbegriff von Team, weil es die member-Rolle genau wie Team füllt und zusätzlich drei Einschränkungen hat, nämlich atmost 4 und atleast 1 leader all leader Woman. Ein Wolfsrudel, das von einer Frau geleitet wird, ist kein Unterbegriff eines Teams.

3.3.4.3 Einige Eigenschaften von Termsubsumtions-Formalismen

Der Klassifikationsalgorithmus kann wahr oder falsch ausgeben für jedes subsumes (t,u) und wenn der Algorithmus wahr ausgibt, dann subsumiert t auch tatsächlich u. Der Algorithmus entscheidet also **korrekt**. Die Subsumtion in zyklusfreien T-Boxen ist **entscheidbar**. Obendrein arbeitet der Algorithmus in polynomialer Zeit über der Länge der beiden zu vergleichenden Terme. Dabei haben wir allerdings einige Konstrukte weggelassen, die in vielen Termsubsumtions-Systemen vorhanden sind: wir haben Zyklen von vornherein ausgeschlossen und insbeson-

dere haben wir die Rollen primitiv gelassen. Es gibt also keine konstruierten Rollen - sobald man diese einführt, wird die Subsumtion **unentscheidbar**. Schon bei dem einfachen Konstrukt `androle` wird die Subsumtion **unvollständig** - oder nicht mehr polynomial. So würde der Algorithmus `falsch` ausgeben im folgenden Fall:

t sei (`atleast` 3 member)

u sei (`and` (`all` (`androle` member programmer) Man)

(`all` (`androle` member scientist) Woman)

(`atleast` 2 (`androle` member programmer))

(`atleast` 2 (`androle` member scientist))

(`disjoint` Man Woman))

Wir sehen nun, daß bei u die Rolle `member` immer mindestens 4 mal vorkommt: zweimal für Männer und zweimal für Frauen. Damit ist die Forderung von t, daß die Rolle mindestens 3 mal vorkommen soll, erfüllt. Also subsumiert t u. Der Algorithmus merkt dies aber nicht und liefert `falsch`. `subsumes` müßte die `disjoint`-Beschränkung berücksichtigen. Im allgemeinen Fall, wenn man nicht auf paarweise Disjunktheit reduziert, muß man dann alle Teilmengen aller Unterrollen untersuchen. In der KI lebt man deshalb mit unvollständiger Subsumtion oder beschränkt den Formalismus. Dies ist das berühmte Abwägen von Brachman, Levesque (1987): Handhabbarkeit oder Ausdrucksfähigkeit eines Repräsentationsformalismus'.

3.3.4.4 Assertionen und hybride Inferenzen

Der Formalismus zur Darstellung der Instanzen, die A-Box, soll nicht mehr Definitionen von Begriffen repräsentieren, sondern Dinge, die unter Begriffe fallen. Die **Syntax** ist einfach (Nebel 1990:65):

<world-description> ::= (<object-description> | <relation-description>)

<object-description> ::= (<atomic-concept> <object>)

<relation-description> ::= (<atomic-role> <object> <object>) |

(<atomic-role> <object> (`atleast` <number>)) |

(<atomic-role> <object> (`atmost` <number>))

Die **Semantik** der Assertionen, Objekt- und Relationsbeschreibungen, kann wieder mit Bezug auf die Menge aller Objekte D angegeben werden. Eine Interpretationsfunktion `int` bildet eine Menge von Objekten auf D ab, eine Menge atomarer Begriffe auf 2^D, eine Menge von Rollen auf $2^{D\times D}$. Eine Interpretation (D,int) **erfüllt** eine Beschreibung δ:

Objektbeschreibung (c o) ist erfüllt, gdw. int(o) ∈ int(c),

Relationsbeschreibung (r o p) ist erfüllt, gdw. (int(o), int(p)) ∈ int(r)

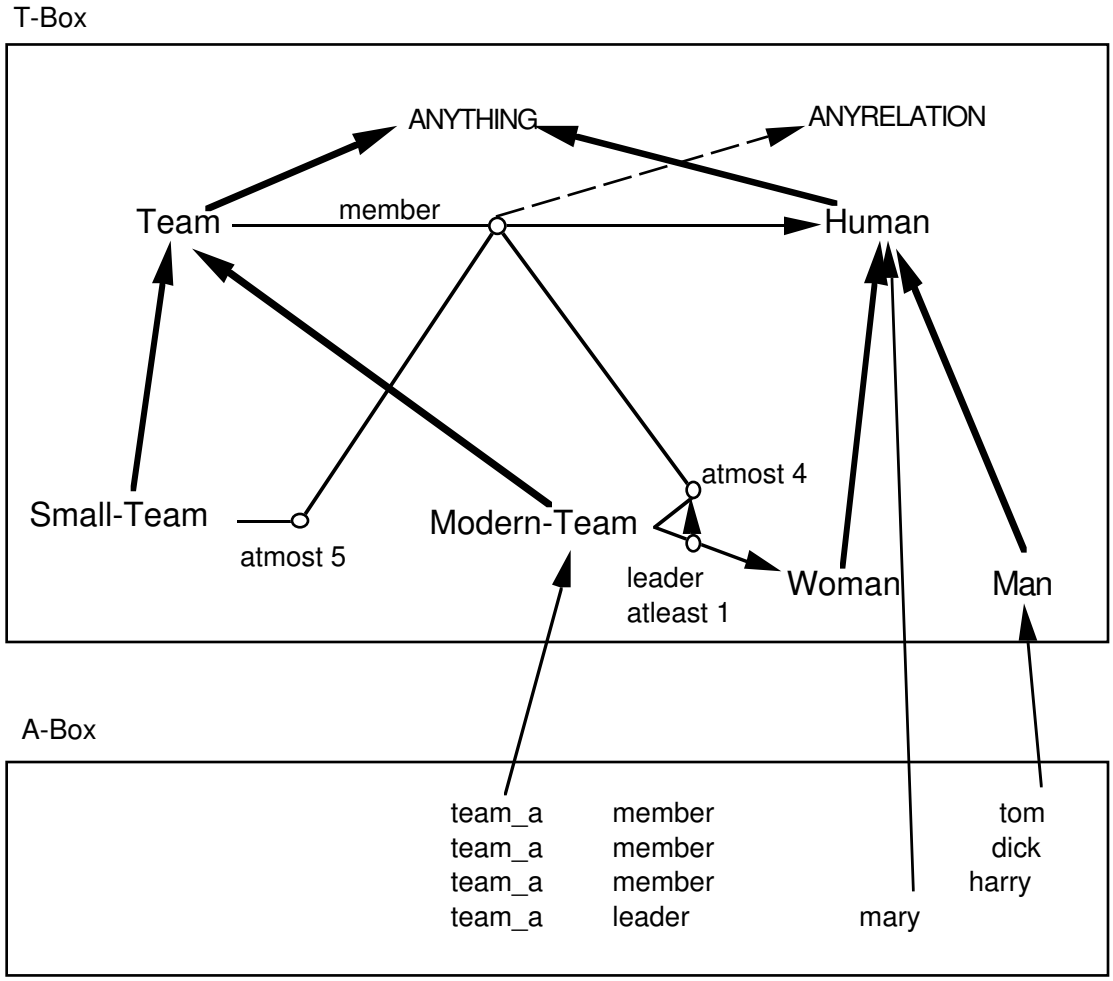
Relationsbeschreibung (r o (`atleast` n)) ist erfüllt, gdw.

$$\text{card}(\ \{X|\ (\text{int}(o),X) \in \text{int}(r)\}\ ) \geq n$$

Relationsbeschreibung (r o (atmost n)) ist erfüllt, gdw.

$$\text{card}(\{X \mid (\text{int}(o),X) \in \text{int}(r)\}) \leq n$$

Wenn eine Interpretation alle Beschreibungen einer A-Box erfüllt, so ist sie ein **Modell** der A-Box. Eine Beschreibung folgt logisch aus einer A-Box, wenn sie in allen Modellen der A-Box erfüllt ist.



Eine A-Box muß nicht alle Angaben enthalten, die gelten. Es kann auch mithilfe der T-Box etwas inferiert werden. Insbesondere können die für einen Begriff gegebenen Eigenschaften an seine Instanzen weitergegeben werden. Wenn zum Beispiel von Mary bekannt ist, daß sie die Leiterin eines modernen Teams ist,

(Modern-Team team_a), (leader team_a mary)

so kann man schließen, daß Mary eine Frau ist. Dies wird auch **hybride Inferenz** genannt, weil sie A-Box und T-Box verbindet. Die einfachste hybride Inferenz stellt fest, ob ein Objekt eine Instanz eines Begriffs ist für eine bestimmte T- und A-Box.

In der Abbildung geben die Pfeile von der A- zur T-Box die ausdrücklich in der A-Box angegebenen Instanz-Beziehungen wieder. Für die Rolle member sind tom, dick und harry die Rollenfüller bzw. Objekte im Wertebereich bzw. Zielobjekte in der A-Box, Human der Rollenfüller bzw. Begriff im Wertebereich in der T-Box. Modern-Team ist der Ursprungsbegriff und team_a das Ursprungsobjekt. Geschrieben wird die Rolle (member Modern-Team Human) bzw. (member team_a tom).

Der **Realisierungsalgorithmus** (*realizer*) stellt fest, von welchem Begriff ein Objekt eine Instanz ist. Dabei wird ein Trick angewandt: für eine A-Box werden alle speziellsten Begriffe berechnet. Sie bekommen einen künstlichen Namen. Die speziellsten Begriffe für eine A-Box decken gerade die Objekt- und Rollenbeschreibungen ab, sind aber meist spezieller als alle schon in der T-Box definierten Begriffe und Rollen. Der Realisierungsalgorithmus propagiert Wertebereichbeschränkungen von Rollen, abstrahiert Beschreibungen eines Objektes oder einer Relation und ruft dann den Klassifikationsalgorithmus auf, der den so gewonnenen speziellsten Begriff in die T-Box einordnet. Der von dem Klassifikationsalgorithmus gefundene Ort in der T-Box ergibt für einen speziellsten Begriff seinen Oberbegriff: das entsprechende A-Box-Objekt ist eine Instanz von diesem Begriff!

Propagierung:

Wenn $\delta = (r\ o\ p)$, dann sammle alle Objektbeschreibungen von o auf, sammle alle Wertebereichseinschränkungen von r und propagiere diese Einschränkungen an p.

Wenn $\delta = (c\ o)$, dann sammle alle Wertebereichseinschränkungen aller Rollen von c, die in der A-Box für o eingetragen sind (r o q), und propagiere diese an das jeweilige q.

Abstraktion:

Für alle Objekte werden alle Begriffe c_i aufgesammelt und konjunktiv verknüpft, die in der Beschreibung $(c_i\ o)$ vorkommen. Wenn der Wertebereich einer Rolle r eingeschränkt ist durch `atleast`, `atmost` oder `all`, wird geprüft, welche Zielobjekte für o bei r eingetragen sind, wieviele es sind und eine entsprechende Wertebereichsbeschränkung für (r o p) wird explizit angegeben.

Klassifikation:

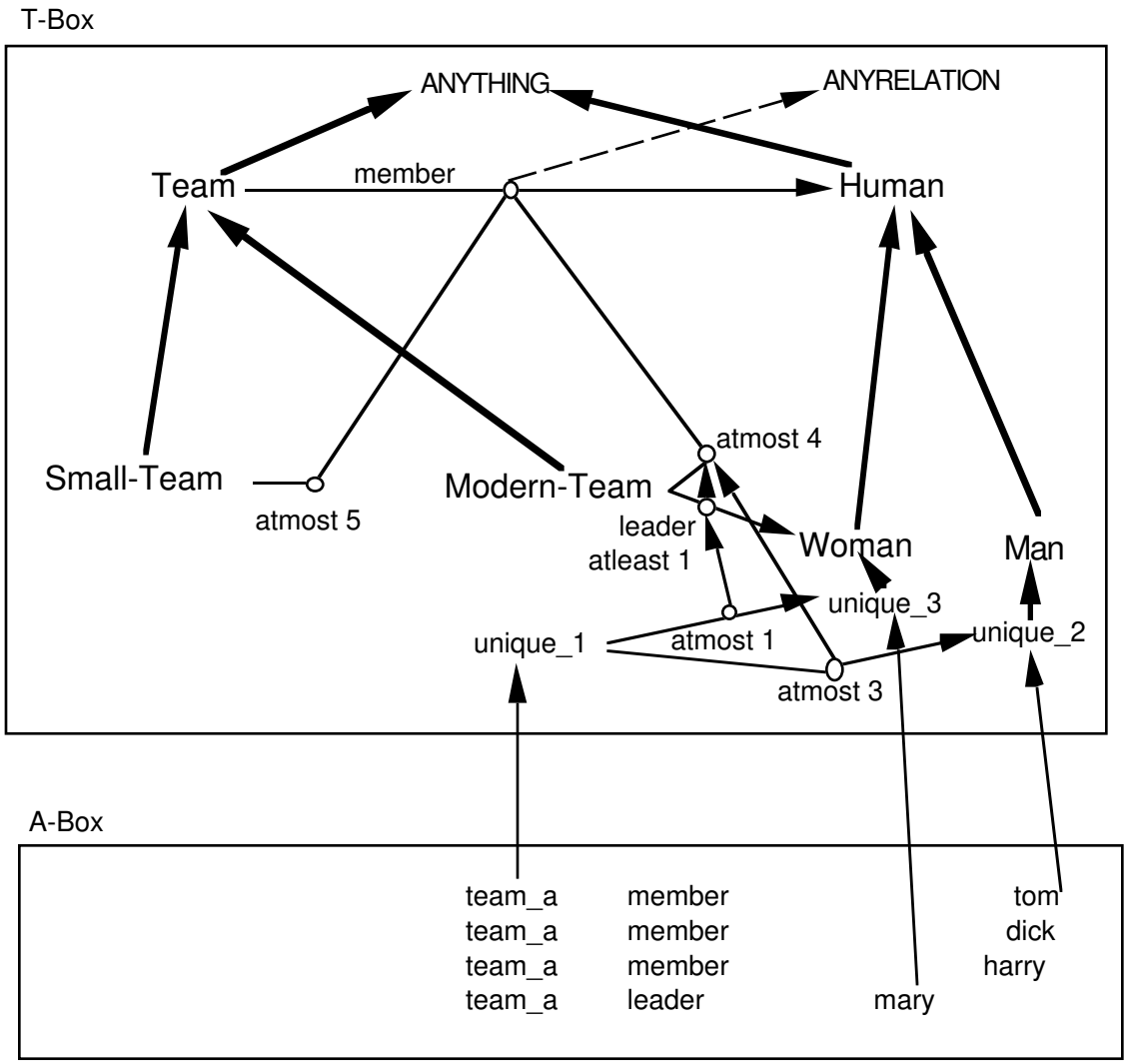
Die durch Abstraktion und Propagierung gewonnene speziellste Definition für eine Objekt- oder Relationsbeschreibung wird in der T-Box klassifiziert.

Für `team_a` wird ein speziellster Begriff definiert und mit dem künstlichen Namen `unique_1` in die T-Box eingetragen:

unique_1 .=. (`and` Modern-Team (`all` member unique_2)(`atmost` 3 member)
(`all` leader unique_3)(`atmost` 1 leader)

(`atleast` 1 leader))

Entsprechend ist für `tom`, `dick`, `harry`, `mary` jeweils ein speziellster Begriff erzeugt worden. unique_2, unique_4 und unique_5 für tom, dick und harry werden vom Klassifikationsalgorithmus als völlig gleiche Definition an dieselbe Stelle der T-Box eingeordnet.



Man kann den Realisierungsalgorithmus als Vorwärtsverkettung von Regeln der T-Box mit einem Grundbeispiel als Ausgangspunkt auffassen. Insofern ist der Aufwand des Algorithmus' durch die Anzahl der Zielobjekte, die Anzahl der Objekte im Wertebereich einer Relationsbeschreibung und die Inferenztiefe gegeben. Deshalb wird oft eine maximale Inferenztiefe als Beschränkung für den Algorithmus angegeben, was natürlich zu seiner Unvollständigkeit führt.

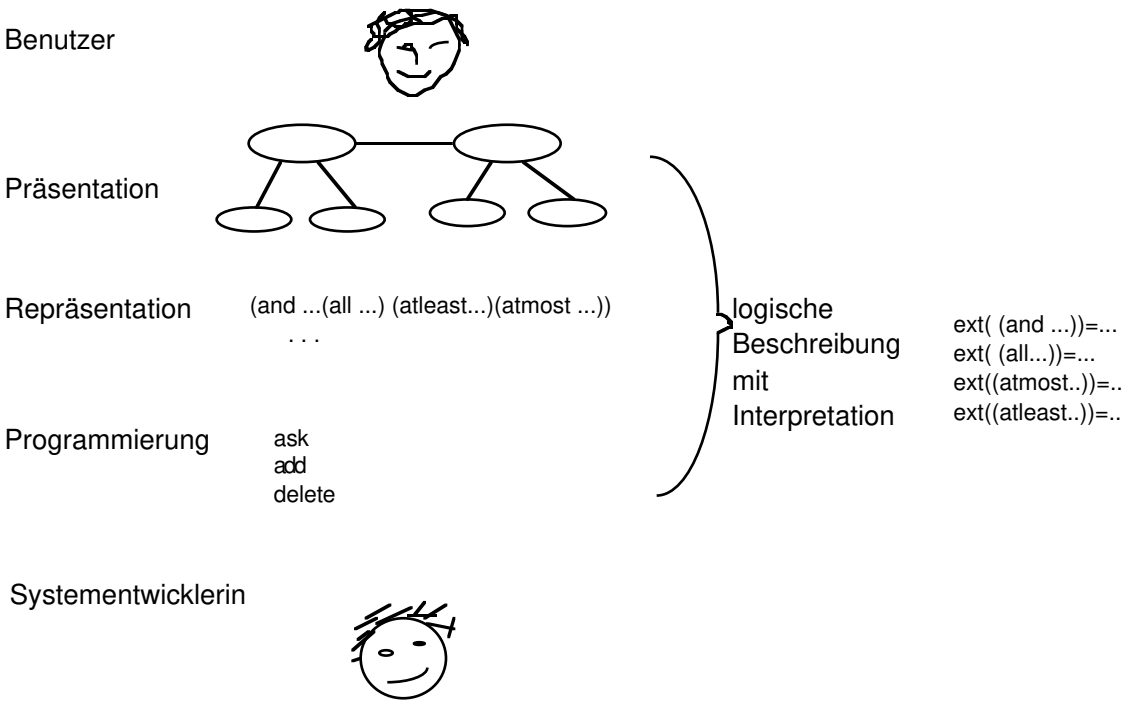
Ein hybrides System, bestehend aus einer T-Box, die Begriffsdefinitionen und Rollen enthält, und einer A-Box, die Beschreibungen von Objekten und Relationen enthält, kann zusammengenommen als Expertensystem eingesetzt werden. Als Standardvariante eines Termsubumtionssystems hat sich CLASSIC (Borgida et al. 1989, Borgida et al. 1992) durchgesetzt.

3.3.5 Zusammenfassung des Lehrstücks

Ausgehend von der einleuchtenden Repräsentation von Begriffen als Knoten und Beziehungen zwischen ihnen als Kanten oder *slots*, begann man das Repertoire von Knoten- und Kantentypen zu untersuchen. Zunächst versuchte man, semantische Primitive zu finden. Das sollten bestimmte Merkmale sein, die zur Modellierung in allen Weltausschnitten verwendet werden können. Es gelang jedoch nicht, einen Kanon von semantischen Primitiven zu finden: einige wie `isa` oder `instance_of` kamen immer wieder vor, andere waren sachbereichsabhängig (wie etwa `dress, present`). Es war auch nicht klar, wie ein Knoten- oder Kantentyp zu verstehen sei. Lauter spezielle Zugriffsprozeduren realisierten ihre operationale

Semantik. Erst als man die epistemische Ebene einführte, ließ sich ein Repertoire aufstellen: das der epistemischen Primitive. Ein von vielen verfolgter Ansatz ist der der Termsubsumtions-Formalismen. Die epistemischen Entitäten sind: Begriffe, Rollen, Wertebereiche von Rollen mit ihren Restriktionen, Schnittmengenbildung bei Begriffen und Rollen. Sie beziehen sich nicht mehr auf Entitäten eines Sachbereichs, sondern auf Entitäten des Definierens von Begriffen. Für jede epistemische Einheit läßt sich eine formale Semantik angeben, die natürlich auch operationalisiert wird durch eine Zugriffsprozedur. Durch die formale Semantik können nun aber sehr unterschiedliche Systeme verglichen werden, ohne daß man ihre Implementation kennen muß. Auch die Implementation selbst ist viel leichter geworden, weil nur für jede epistemische Entität eine Prozedur für das Hinzufügen, eine für das Löschen und eine für eine Anfrage geschrieben werden muß. Obendrein können formale Eigenschaften für Termsubsumtions-Formalismen angegeben werden: die Klassifikation ist korrekt und - wenn keine Zyklen auftreten können - entscheidbar. Wenn auch nur die Schnittbildung bei Rollen zugelassen wird, ist die Subsumtion unvollständig. Damit wissen wir, daß wir nicht jeder negativen Antwort des Systems trauen dürfen. Wenn keine Rollenkomposition zugelassen wird, ist der Aufwand polynomial. Damit wissen wir zum Beispiel, daß eine exponentielle Implementation hinter dem Stand der Kunst herhinkt.

Das folgende Bild veranschaulicht die Ebenen eines Wissensrepräsentationssystems. Dem Benutzer wird von der Benutzerschnittstelle ein bestimmtes Format des Wissensrepräsentationsformalismus' präsentiert. Zum Beispiel können graphische Objekte oder textuelle Einheiten dargestellt und vom Benutzer manipuliert werden. Diesem Präsentationsformat (mit Ellipsen und Pfeilen etwa) liegt der Repräsentationsformalismus (mit Klassen von Objekten und Relationen) zugrunde. Der wird realisiert mithilfe eines Programmes, das die epistemischen Einheiten des Formalismus verarbeitet. Zum Beispiel werden neue Einheiten eingetragen, wobei Integritätsbedingungen und vielleicht auch die Konsistenz mit vorhandenem Wissen geprüft werden, oder es wird nach bestimmten Einheiten im Wissen gesucht, wobei sie auch gefolgert werden können. Das Verarbeitungsprogramm gibt an, was eine Einheit des Formalismus' bedeuten soll, indem es die Verarbeitung der Einheit vorschreibt. Das Verarbeitungsprogramm wird seinerseits auf einer bestimmten Maschine mit einem bestimmten Betriebssystem realisiert. Dies ist die operationale Semantik des Repräsentationsformalismus'. Der Repräsentationsformalismus erhält eine logische Beschreibung, die angibt, was die Einheiten bedeuten sollen. Dies ist die formale Semantik des Repräsentationsformalismus'. Sie soll lesbarer sein als das Verarbeitungsprogramm und unabhängig von maschinenspezifischen Details. Die logische Beschreibung ist auf ein logisches Kalkül (also Regeln wie etwa die Schnittregel und Axiome) gegründet.



3.3.5.1 Literatur

Baader, Franz (1990): Terminological cycles in KL-ONE-based knowledge representation languages. in: Proceedings of the 8th National Conference of the American Association for Artificial Intelligence, 1990.

**Borgida, A., Brachman, R., McGuinness, D.L., Resnick, L. (1989): CLASSIC -- A structural data model for objects, in: Procs. of SIGMOD-89, Portland.

Borgida, A., Patel-Schneider, P.F. (1992): A semantics and complete algorithm for subsumption in the CLASSIC description logic, AT&T Technical Report.

*Brachman, Ron J. (1977): What's in a Concept - Structural Foundations for Semantic Networks, in: *Int. Journal of Man-Machine Studies*, 9, 127-152

Brachman, Ron J. (1979): On the Epistemological Status of Semantic Networks, in: Findler(ed): *Associative Networks - Representation and Use of Knowledge by Computers*, New York: Academic Press, 3-50

Brachman, Ron J., Levesque, Hector (1987): The Tractability of Subsumption in Frame-based Description Languages, in: Procs. 4th AAAI-84, Austin, Texas, 1987

**Brachman,Ron, Levesque, Hector (eds): *Readings in Knowledge Representation*, Los Altos: Morgan Kaufmann, 1985.

Hayes, Patrick J. (1977): In Defence of Logic, in: Proceedings of the fifth International Joint Conference on Artificial Intelligence, 1977.

McDermott, Drew V. (1978): Artificial Intelligence Meets Natural Stupidity, in: *SIGART Newsletter*, 57.

Minsky, Marvin (1981): A Framework for Representing Knowledge, in: Haugeland (ed): *Mind Design*, MIT Press, 1981

*Nebel, Bernhard (1990): Reasoning and Revision in Hybrid Representation Systems, Berlin, New York: Springer

Schank, Roger (1973): Identification of Conceptualization Underlying Natural Language, in: Schank, Colby (eds): *Computer Models of Thought and Language*, San Francisco: Freeman A

von Luck, Kai, Nebel, Bernhard, Peltason, Christof , Schmiedel, Albrecht (1987): The Anatomy of the BACK System, TU Berlin, Projektgruppe KIT, KIT-Report 41, Januar 1987.

3.4 Produktionensysteme

Für ein Produktionensystem der Künstlichen Intelligenz gibt es drei Herleitungen:

- dieser Formalismus sollte ein Gedächtnismodell operationalisieren,
- es ist die konsequente Weiterentwicklung der Suche über UND-ODER-Graphen für praktische Anwendungen,
- dieser Formalismus sollte den allgemeinen Interpreter von den speziellen Inhalten eines Sachbereichs trennen.

Wir haben hier also wieder alle drei Bestimmungen der KI zusammen: das kognitive Verhalten des Menschen zu beschreiben, Programme menschengerechter zu machen und expliziter und verständlicher zu operationalisieren.

3.4.1 Produktionensysteme als Gedächtnisbeschreibung

Eine Reihe von Studien in den 60er Jahren führte dazu, daß man beim Menschen ein Kurzzeitgedächtnis mit einer festen, kleinen Speichergröße und ein Langzeitgedächtnis mit einer flexiblen, großen Speichergröße annahm. Die Speichergröße des Kurzzeitgedächtnis wurde mit 7 Einheiten (chunks) angegeben (Miller 1956). Was einer Einheit entspricht, ist dabei unterschiedlich. Wenn wir Lesen lernen, ist für uns ein Buchstabe eine Einheit und wir können uns nur wenige (etwa 7) Buchstaben merken. Wenn wir schon lesen können, ist ein ganzes Wort für uns eine Einheit und wir können uns mehrere (etwa 7) Wörter merken. Wir können stehende Redewendungen wiederum als Einheiten, die aus Wörtern aufgebaut sind, begreifen. Wir können uns dann 7 Redewendungen merken. Sie können dieses Prinzip leicht bei sich selbst beobachten. Die folgenden Wörter in arabischen Buchstaben sollen Sie wiedergeben:

Geht nicht? Nun, nehmen wir lediglich das erste Wort:

Schon besser? Jetzt verrate ich Ihnen, daß - von rechts nach links - die 9 Buchstaben bedeuten:

Ka Ta R I Na M U R I K.

So. Jetzt zeige ich die arabischen Buchstaben noch einmal mit dieser Lesung versehen. Können Sie die Wörter wiedergeben? Diese Folge dürfte für Sie maximal 6 Einheiten ausmachen, denn Katharina ist ein ganz normaler Name. Psychologen haben das Gedächtnis durch Wiederholungsexperimente mit sinnhaftem und sinnlosem Material untersucht: den Versuchspersonen wurden beispielsweise Silben präsentiert, die sie danach wiederholen sollten. Dabei traten insbesondere zwei Effekte auf: die Menge der gelernten Silben hängt von den kognitiven Ein-

heiten (chunks) ab (bekannte Silben brauchen weniger Zeit/Platz) und die ersten und die letzten Silben werden am besten erinnert (Positionseffekt).

Natürlich wissen wir mehr, als mit 7 Einheiten zu kodieren wäre. Das Kurzzeitgedächtnis ist nur der Inhalt, über dem aktuell Operationen ausgeführt werden. Das Langzeitgedächtnis umfaßt mehr. Man versuchte, das Langzeitgedächtnis als Index und Gedächtnisinhalt aufzufassen, wobei der Index die Zugriffspfade zu den Inhalten darstellt. Lernen wäre dann unter anderem der Aufbau eines solchen Index und das Verwenden eines Index, um eine Antwort auf eine Frage zu finden.

Feigenbaum entwickelte in seiner Dissertation ein operationales Modell des Gedächtnisses. Das Modell, EPAM (elementary perceiver and memorizer), besteht in einem Baum, der durchsucht wird, um einen Gedächtnisinhalt zu finden (Feigenbaum, Simon 1963). Dabei sind die Kanten Tests und deren Nachfolgeknoten sind die Elemente, die noch im Zugriff sind, wenn der Test bestanden wurde. Die Blätter sind die aufzufindenden Elemente, also chunks. Zum Beispiel könnte ein Wort so gefunden werden: zuerst nimmt man den ersten Buchstaben und findet damit einen Knoten, von dem aus nur noch die Wörter mit diesem Anfangsbuchstaben erreichbar sind; von diesem Knoten aus wählt man die Kante mit dem letzten Buchstaben des Wortes; von dem so gefundenen Knoten wählt man die Kante mit dem ersten der mittleren Buchstaben und so fort, bis man bei dem gesuchten Wort angekommen ist. Der Baum ist ein Diskriminationsnetz. Wenn man dieses einfache Modell zur Beschreibung der Effekte beim Erinnern etwas weiterführt, kann man auch Handlungen oder Problemlösungen beschreiben. Die Blätter des Diskriminationsnetzes sind dann nicht mehr einfach nur Einheiten, sondern Bedingungs-Handlungspaare, die in bestimmten Situationen relevant sind. Die Bedingungen geben an, wann die Handlung auszuführen ist. Eine Regel, die eine Menge von Bedingungen mit einer Handlung verknüpft, ist eine **Produktionenregel**. Eingeführt wurde der Formalismus der Produktionen von Post (1943). Sein Produktionensystem besteht aus

A: einem Alphabet bestehend aus den disjunkten Mengen terminaler und nonterminaler Symbole,

Axiom: einem ausgezeichneten Wort, mit dem die Ableitungen beginnen,

Produktionen: W --> V mit W, V $\in$ A* (hier ist kein Suchalgorithmus gemeint, sondern die Konkatenationen von Symbolen aus A). Eine Produktion drückt aus, daß das Teilwort W durch das Teilwort V ersetzt werden darf.

Das Ergebnis einer Produktion wird einem Speicher hinzugefügt, in dem anfangs nur das Axiom ist.

Die Psychologen griffen den Formalismus für ihre Modellierung wieder auf. Die Produktionen werden nun geschrieben:

IF <Bedingung$_1$,...,Bedingung$_n$> THEN <Handlung>

Ein **Produktionensystem** besteht aus einer Menge von Produktionenregeln, einem Kurzzeitgedächtnis (dem impliziten Speicher bei Post) und einem Interpreter. Das Kurzzeitgedächtnis gibt die aktuelle Situation an, weswegen es auch "Kontext" genannt wird. Die Benennung des Kurzzeitgedächtnisses als "Datenbasis" hat zu fürchterlichen Mißverständnissen geführt, da Datenbasen üblicherweise groß sind und - psychologisch betrachtet - eher dem Langzeitgedächtnis nahe kämen, auf gar keinen Fall dem Kurzzeitgedächtnis. Tatsächlich ist das Kurzzeitgedächtnis lediglich eine Repräsentation der bisher abgeleiteten Inhalte. Gegen diese aktuelle Situation werden die Bedingungen der Produktionen-

regeln abgeprüft. Die Aktionen der Produktionenregeln verändern den Inhalt des Kurzzeitgedächtnisses. Dadurch werden dann andere Regeln anwendbar, d.h. ihre Bedingungen sind dann vielleicht erfüllt. Der Interpreter hat die Aufgabe zu bestimmen, welche Regel als nächste angewandt werden soll. Es kann nur eine sein, deren Bedingungsteil vom Inhalt des Kurzzeitgedächtnisses erfüllt wird. Ob man nur die erste anwendbare Regel nimmt oder alle oder welche man aus den anwendbaren auswählt, ist die Interpreterstrategie. Die Ausführung der Regel besteht darin, den Inhalt des Kurzzeitgedächtnisses zu verändern.

Was die Psychologen an den Produktionensystemen so interessant fanden, war, daß Regeln erlernbar aussehen (Simon 1978). Man kann sich leichter vorstellen, daß Lernen unter anderem der Erwerb von zusätzlichen Regeln ist, als man sich vorstellen kann, daß ein ganzes Pascal-Programm - das ja ununterscheidbar Daten, Kontrolle, Veränderung der Situation enthält - durch Lernen erworben und erweitert wird. Die Trennung der Bestandteile eines klassischen Programms in einen allgemeinen und gleichbleibenden Interpreter, ein dynamisches Kurzzeitgedächtnis und eine Menge gleichförmiger, kleiner Einheiten, die Situation und Handlung in Verbindung bringen, ermöglicht die - auch automatische - Veränderung einer Menge (kognitiver) Einheiten. Um das Verhalten des gesamten Systems zu verändern, braucht man nur eine Regel hinzuzufügen oder eine Regel zu löschen oder eine Regel zu modifizieren. Oder, anders ausgedrückt, wenn sich die Regelmenge ändert, ändert sich auch das Verhalten des Systems. Die Regelmenge wurde dann als "Wissen" bezeichnet, weil sie im Gegensatz zu Eingabedaten in ein Pascal-Programm, die ja sicherlich auch das Verhalten des Programms bestimmen, auch Teile dessen enthalten, was im Pascal-Programm stehen würde. Natürlich ist die Wahl des Begriffs "Wissen" sehr problematisch (Morik 1991). Hier sollte nur skizziert werden, aus welchen Überlegungen heraus man überhaupt auf die Idee kam, Produktionenregeln als Wissen zu bezeichnen. Dies ist ohne den Hintergrund, daß dieser Formalismus als Modell des menschlichen Gedächtnis interpretiert wurde, wohl nicht zu verstehen.

3.4.2 Produktionensysteme technisch

Gerade der Ingenieursansatz der KI hat sich mit Produktionensystemen beschäftigt. An einem kleinen Beispiel soll ein Produktionensystem "bei der Arbeit" gezeigt werden. Wir nehmen folgende Menge von Regeln an:

```
R1:  IF ok(Geraet) & verbunden(Geraet, Sicherung)
     THEN intakt(Sicherung)

R2:  IF eingeschaltet(Geraet) & arbeitet(Geraet)
     THEN ok(Geraet)

R3:  IF eingeschaltet(Geraet) & tutnicht(Geraet)
     THEN gestoert(Geraet)

R4:  IF verbunden(Geraet1, Sicherung) & verbunden(Geraet2, Sicherung)
     THEN gleiche_sicherung(Geraet1, Geraet2,Sicherung)

R5:  IF gestoert(Geraet) & verbunden(Geraet,Sicherung)
     THEN verdacht(Geraet,Sicherung)

R6:  IF gleiche_sicherung(Geraet1, Geraet2,Sicherung) &
        gestoert(Geraet1) & gestoert(Geraet2)
     THEN defekt(Sicherung)

R7:  IF gleiche_sicherung(Geraet1, Geraet2, Sicherung) &
        gestoert(Geraet1) & ok(Geraet2)
     THEN andere_stoerung(Geraet1)
```

Nehmen wir an, das Kurzzeitgedächtnis bzw. der Kontext enthielte zunächst

```
eingeschaltet(lampe1). tutnicht(lampe1).
eingeschaltet(lampe2). tutnicht(lampe2).
verbunden(lampe1,sicherung1). verbunden(lampe2,sicherung1).
```

Nehmen wir weiterhin einen einfachen Interpreter an. Jeder Interpreter eines Produktionensystems durchläuft den Zyklus: Abgleich (match), Konfliktauflösung (conflict resolution) und Anwendung (act).

Eine einfache Form sieht etwa so aus:

Abgleich: Finde alle Regeln, deren Bedingungsteil sich mit dem Kontext abgleichen läßt;

Markiere diese Regeln als "anwendbar";

Wenn es keine anwendbaren Regeln gibt, gib den Kontext aus und terminiere;

Konfliktauflösung: Wähle die erste anwendbare Regel aus; entferne die Markierung von allen anderen anwendbaren Regeln;

Anwendung: führe den Handlungsteil der ausgewählten Regel aus, indem die neue Aussage in den Kontext geschrieben wird und die den Bedingungen entsprechenden Aussagen aus dem Kontext gelöscht werden;

entferne die Markierung von der Regel;

In diesem Beispiel für einen Interpreter wird vom Kontext ausgehend die Menge der anwendbaren Regeln durch den Abgleich mit den Bedingungen bestimmt. Diesen Modus eines Interpreters nennt man **Vorwärtsverkettung**. Dies entspricht dem Vorgehen bei dem Postschen Produktionssystem.

Im ersten Zyklus des angegebenen Interpreters sind die Regeln R3 und R4 anwendbar. Die Konfliktauflösung wählt R3 aus. Die Regelanwendung fügt

```
gestoert(lampe1)
```

dem Kontext hinzu und löscht aus dem Kontext:

```
eingeschaltet(lampe1), tutnicht(lampe1)
```

Im nächsten Zyklus sind die Regeln R3, R4 und R5 anwendbar. Es wird wieder R3 ausgewählt. Dem Kontext wird hinzugefügt:

```
gestoert(lampe2)
```

Gelöscht wird aus dem Kontext:

```
eingeschaltet(lampe2), tutnicht(lampe2)
```

Im nächsten Zyklus sind die Regeln R4 und R5 anwendbar. Es wird R4 ausgewählt, so daß in den Kontext geschrieben wird:

```
gleiche_sicherung(lampe1, lampe2,sicherung1)
```

und gelöscht wird:

```
verbunden(lampe1, sicherung1), verbunden(lampe2, sicherung1)
```

Nun ist nur noch R6 anwendbar. Der Kontext wird gelöscht, und

```
defekt(sicherung1)
```

wird hineingeschrieben. Dies ist nun die einzige Aussage, die im Kontext steht. Im nächsten Zyklus ist keine Regel anwendbar, der Inhalt des Kontextes wird ausgegeben und der Lauf des Produktionensystems ist beendet. Die Regeln sind so geschrieben, daß jeder mögliche letzte Kontextinhalt sinnvoll ist. Man kann aber auch bestimmte Aussagen als Lösungen vorsehen. In diesem Beispiel könnten

```
intakt(Sicherung), defekt(Sicherung)
```

als mögliche Lösungen ausgewählt werden. Oder zusätzlich könnte auch

```
andere_stoerung(Geraet)
```

eine den Anwender befriedigende Lösung sein. Wenn das Produktionensystem dann bei

```
verdacht(Geraet, Sicherung)
```

terminiert, wäre der Anwender nicht zufrieden. Man könnte diese Lösung dann als Sackgasse betrachten.

Wir können die **Vorwärtsverkettung** für den aussagenlogischen Fall in Prolog etwa folgendermaßen realisieren.

```
:- ensure_loaded(library(basics)).

:- dynamic(context/1).

closure :-
   forward_inference(Konklusion), % wenn noch neue Fakten ableitar
   assert(context(Konklusion)),   % fuege sie zum Kontext hinzu.
   closure.

closure.

forward_inference(Attribut=Wert) :-
   rule(if:Praemissen, then: [Attribut=Wert]),
   \+ context ([Attribut= _]),   % nur wenn noch nicht bekannt und
   alltrue(Praemissen).          % alle Praemissen erfuellt sind.

alltrue([]).

alltrue([X|Praemissen]) :-
   context(X),
   alltrue(Praemissen).
```

Der andere Modus ist der der **Rückwärtsverkettung**. Dort wird von einem Ziel ausgehend die Menge der anwendbaren Regeln durch den Abgleich mit dem Handlungsteil bestimmt. Wenn eine Bedingung einer Regel nicht erfüllt ist, so wird diese Bedingung als neues Ziel gesetzt und als nächstes nach einer Regel gesucht, die diese Bedingung wahr macht. Dies entspricht dem Vorgehen von Prolog und nicht dem der Postschen Produktionssysteme. Wir können auch in Prolog die Rückwärtsverkettung für Produktionsregeln programmieren.

```
backward_inference(Konklusion) :-
   context(Konklusion).              % Wenn in Kontext, dann erfüllt

backward_inference(Konklusion) :-
   rule(if: Praemissen, then: [Konklusion]),
   derive_all(Praemissen).           % Prämissen werden zu neuen Zielen

backward_inference (Attribut=Wert) :-
   ask(Attribut, Wert).              % Frage den Benutzer

derive_all([]).
```

```
derive_all([X|Praemissen]) :-
   backward_inference(X),
   derive_all(Praemissen).

ask(Attribut,Wert) :-
   can_be_asked(Attribut,Prompt, Werte),
   \+ context(Attribut= _),
   write(Attribut), write(Prompt), write(Werte), write( ´  ´),
   read(Wert1),
   assert(context(Attribut=Wert1)),
   Wert = Wert1.
```

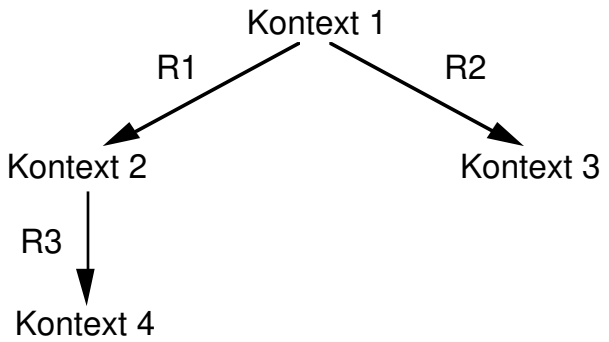
3.4.2.1 Interpreterstrategien

Bei Produktionensystemen ist die Reihenfolge der Regeln wichtig - was dem Gedanken der Trennung von Kontrolle (im Interpreter) und rein beschreibenden Einheiten (Regeln) widerspricht. Wenn eine Regel anewandt wurde, so wird ihre Konklusion (bei der Vorwärtsinferenz) in den Kontext eingetragen und nicht mehr zurückgezogen. Eine Interpreterstrategie ist **zerstörerisch** (irrevocable), weil eine einmal getroffene Entscheidung für die Anwendung einer bestimmten Regel nicht zurückgezogen werden kann. Die alternativen anwendbaren Regeln sowie der Kontextinhalt zum Zeitpunkt der Entscheidung werden nicht aufbewahrt. Für den Menschen, der die Regeln schreibt, bedeutet dies, daß er auf die Unabhängigkeit der Bedingungsteile von Regeln genau achten und die Reihenfolge der Regeln günstig wählen muß.

Eine **tentative** Strategie wählt zwar auch eine Regel zur Anwendung aus, ermöglicht aber, daß auch später noch Alternativen betrachtet werden können. Das Rückziehverfahren kehrt an den Punkt der Auswahl (Rückzugspunkt, englisch: backtracking point) zurück und wählt eine alternative Regel aus. Eine andere tentative Strategie ist an Graphen orientiert. Sie verfolgt die Auswirkungen von Regelanwendungen simultan. Suchverfahren, wie sie in Kapitel 2 beschrieben wurden, können verwendet werden.

Das **Rückziehverfahren** erfordert, daß die Kontextinhalte zu der jeweiligen Regelauswahl etwa in Form einer Liste gespeichert werden. Anfangs ist nur der gegebene Inhalt des Kontextes Element dieser Liste. Der von einer Regelanwendung produzierte Kontextinhalt wird vorn an die Liste angehängt. Die angewandte Regel wird notiert. Wenn von einer ausgewählten Regel ein Kontextinhalt produziert würde, der bereits in der Liste vorkommt, so handelt es sich um eine zyklische Regelfolge. Dies kann die Konfliktauflösung verwenden, um eine andere Regel auszuwählen. Wenn eine Regelfolge nicht zu einer Lösung führt, wird in der Kontextliste zurückgegangen und für den dort festgehaltenen nächsten Kontextinhalt nach einer alternativen anwendbaren Regel gesucht. Gibt es keine andere anwendbare Regel, wird in der Kontextliste weiter zum nächsten Kontextinhalt zurückgegangen und dort nach alternativen Regeln gesucht. Gibt es eine alternative anwendbare Regel, wird die Kontextliste bis zu diesem Kontextinhalt gelöscht. Um wirklich immer auf alternative Regeln zu stoßen, werden aus der Regelliste die bereits angewandten Regeln zu dem Kontext gelöscht.

Wir können uns das Rückziehverfahren an einem Graph klarmachen, dessen Knoten Kontextinhalte und dessen Kanten Regelanwendungen sind:



Das Rückziehverfahren könnte z.B. von Kontext 4 zu Kontext 2 zurückgehen und von da aus zu Kontext 1, um dann die alternative Regel R2 auszuwählen. Kontext 2 und 4 wären dann weggelöscht. Da wir jetzt aber schon die Regelauswahl und die Kontextinhalte als Graph aufgefaßt haben, können wir auch alle bereits behandelten Suchverfahren anwenden und damit flexiblere Interpreterstrategien erstellen.

3.4.2.2 Abgleich

Bei der Besprechung der Interpreterstrategien haben wir vorausgesetzt, daß Regeln als "anwendbar" markiert werden. In der Prolog-Implementierung haben wir die Unifikation und die Gleichheit `Attribut=Wert` verwendet. Der Abgleich zwischen Kontext und Regelbedingungen ist aber ein für sich interessanter Prozeß. Deshalb soll er in diesem Abschnitt behandelt werden.

Der Abgleich einer Bedingung kann bereits aufwendig sein. So kann zum Beispiel ein Intervall von Werten statt nur ein einziger Wert von einer Regel abgedeckt werden. Der konsistente Abgleich mehrerer Bedingungen fügt einen weiteren Aufwand hinzu. Zum Beispiel gleicht die folgende Bedingung einer Regel

```
verbunden(Geraet, Sicherung)
```

die folgenden möglichen Kontextinhalte ab:

```
verbunden(lampe1, sicherung1).

verbunden(lampe2, sicherung1).

verbunden(radio1, sicherung2).
```

Wenn jetzt wie in Regel R4 zwei verschiedene Variablenbindungen für `Geraet` und dieselbe für `Sicherung` gesucht wird, so kommen nur noch die ersten beiden Kontextinhalte zum Abgleich für beide Bedingungen von R4 in Frage. Der Abgleich verschiedener Bedingungen einer Regel ist nicht unabhängig voneinander.

Der Abgleich, den ein Interpreter vornimmt, geht alle linken Seiten von allen Regeln durch und gleicht sie mit allen Elementen des Kontextes ab. Das Ergebnis ist eine Menge von geordneten Paaren

```
(Regelnummer, Liste der abgeglichen Kontextinhalte).
```

Diese Menge anwendbarer Regeln, **Konfliktmenge** genannt, wird von der Konfliktauflösung verwendet. Am Anfang unseres Beispiels sähe die Konfliktmenge so aus:

```
(R3,  (eingeschaltet(Geraet), tutnicht(Geraet),
            Geraet/lampe1, Geraet/lampe2))
```

```
(R4,  (verbunden(Geraet1,Sicherung), verbunden(Geraet2,Sicherung),
            Geraet1/lampe1, Sicherung/sicherung1, Geraet2/lampe2))
```

Dabei werden zunächst die Bedingungen und dann die abgeglichenen Variablen mit den Konstanten notiert. Bei alternativen Abgleichen wie zum Beispiel bei Regel R3 werden die Alternativen durch ";" getrennt. Ein solches Vorgehen bei jedem Zyklus des Interpreters anzuwenden, ist zu aufwendig. Tatsächlich wurde festgestellt, daß ein Produktionensystem 90% seiner Zeit mit Abgleichen verbringen kann (Forgy 1982). Deshalb speichert man die Konfliktmenge. Wenn ein Kontextinhalt gelöscht wird, muß man nicht mehr alle Regeln durchgehen, sondern lediglich eine oder mehrere Paare der Konfliktmenge löschen. Wenn ein Kontextinhalt neu hinzukommt, müssen die Regeln gefunden werden, deren Nummern mit diesem Kontextinhalt zusammen in die Konfliktmenge aufgenommen werden. Aber auch das ist noch zu aufwendig. Deshalb wurde die Regelkompilierung eingeführt. Der am meisten verwendete Algorithmus zur Regelkompilierung, der auch die Konfliktmenge erstellt und wartet, ist **Rete** (Forgy 1982).

3.4.2.3 Rete

Das Rete-Verfahren optimiert den Abgleich von Bedingungen und Kontextinhalten, indem es vermeidet,

- in jedem Zyklus den Kontext durchzugehen und
- in jedem Zyklus die Regelmenge durchzugehen.

Stattdessen werden nur Änderungen des Kontextes und diese nur für Teile der Regelmenge (die nämlich von der Änderung betroffen sind) gewartet.

Das Rete-Verfahren überführt die linken Seiten aller Regeln in ein Netzwerk. Und zwar werden Merkmale von Bedingungen, die bei Kontextinhalten abgeprüft werden, aufgestellt. Solche Merkmale sind abhängig von der verwendeten Sprache, in der die Bedingungen formuliert werden. Für die zweite Bedingung von R1 in unserem Beispiel können wir die folgenden Merkmale, die für einen Kontextinhalt gelten oder nicht gelten, aufstellen:

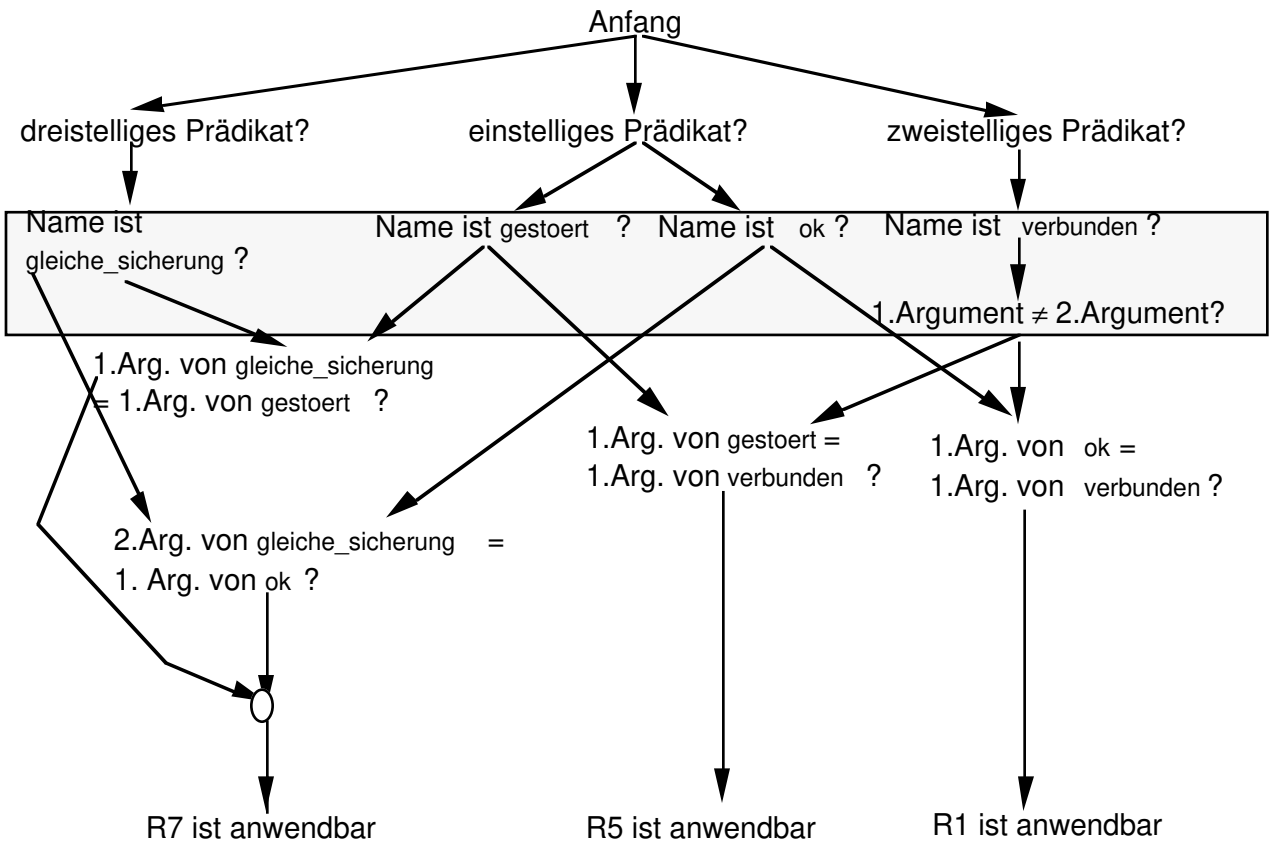
- Stelligkeit des Prädikats ist 2
- Name des Prädikats ist verbunden
- 1. Argument muß ungleich dem 2. Argument sein
- 1. Argument muß ...

Die Bedingungen für die einzelnen Argumente sind in unserem Falle nicht gegeben. Es könnte aber auch in einer Regel im Bedingungsteil eine Konstante auftreten. Dann würde ein Merkmal sein, daß genau diese Konstante an genau dieser Argumentstelle im Kontextinhalt vorkommen muß. Die Regelkompilierung erstellt für jede Eigenschaft einen Knoten und verbindet die Knoten. Knoten für das Abprüfen einer Eigenschaft heißen **Einer-Knoten**. Andere Merkmale lassen sich aufstellen, die mehrere Bedingungen einer Regel und damit mehrere Kontextinhalte verknüpfen. Für die Bedingungen von R1:

- Gleichheit des 1. Arguments von ok und des 1. Arguments von verbunden

Auch für diese Eigenschaft wird ein Knoten erstellt. Diese verbindenden Knoten nehmen zwei Kanten auf, verbinden also Pfade. Diese Knoten heißen **Verbindungsknoten**.

In dieser Weise werden alle Bedingungen aller Regeln einmal durchgegangen. Wenn dieselbe Bedingung in verschiedenen Regeln vorkommt, wird derselbe Knoten mehrfach verwendet. Wenn dasselbe Prädikat in derselben Regel zweimal vorkommt, muß dafür ein neuer Knoten eingeführt werden. Als **terminale Knoten** fügt Rete jeweils für eine Sequenz von Tests (ein Pfad) die Menge der anwendbaren Regeln hinzu. Ein Anfangsknoten wird vorangestellt. Weitere Knotentypen können für negierte Bedingungen und für mehrfaches Auftreten derselben Variable oder Konstante in einem Prädikat eingeführt werden. Das kleine Netzwerk für R1, R5 und R7 (linke Seiten) sieht so aus[22]:



Dieses Netzwerk, das fast so groß wird wie die Regelmenge (im schlimmsten Fall so groß wie die Menge der Bedingungsteile aller Regeln), lohnt sich nur, wenn Änderungen des Kontextes durch dieses Netzwerk propagiert werden, so daß die aktuelle Konfliktmenge schneller als durch Prüfen aller Regeln berechnet wird.

Kontextinhalte werden mit ihren Merkmalen in das Netzwerk eingegeben. Sie erhalten ein '+', wenn sie dem Kontext hinzugefügt wurden, und ein '-', wenn sie gelöscht wurden. Die so annotierten Merkmale von Kontextinhalten heißen **Marken**. Eine Marke wird für alle Nachfolgeknoten des Anfangsknotens kopiert. Die Einerknoten geben eine Marke, die ihren Test besteht, an alle Nachfolger weiter. Die Verbindungsknoten bekommen verschiedene Marken, die sie zu einer komplexeren Marke kombinieren. Wenn die aufgenommene Marke ein positives Vorzeichen hatte, wird sie beim Verbindungsknoten in einem Zwischenspeicher abgelegt. Wenn die aufgenommene Marke ein negatives Vorzeichen hatte, wird

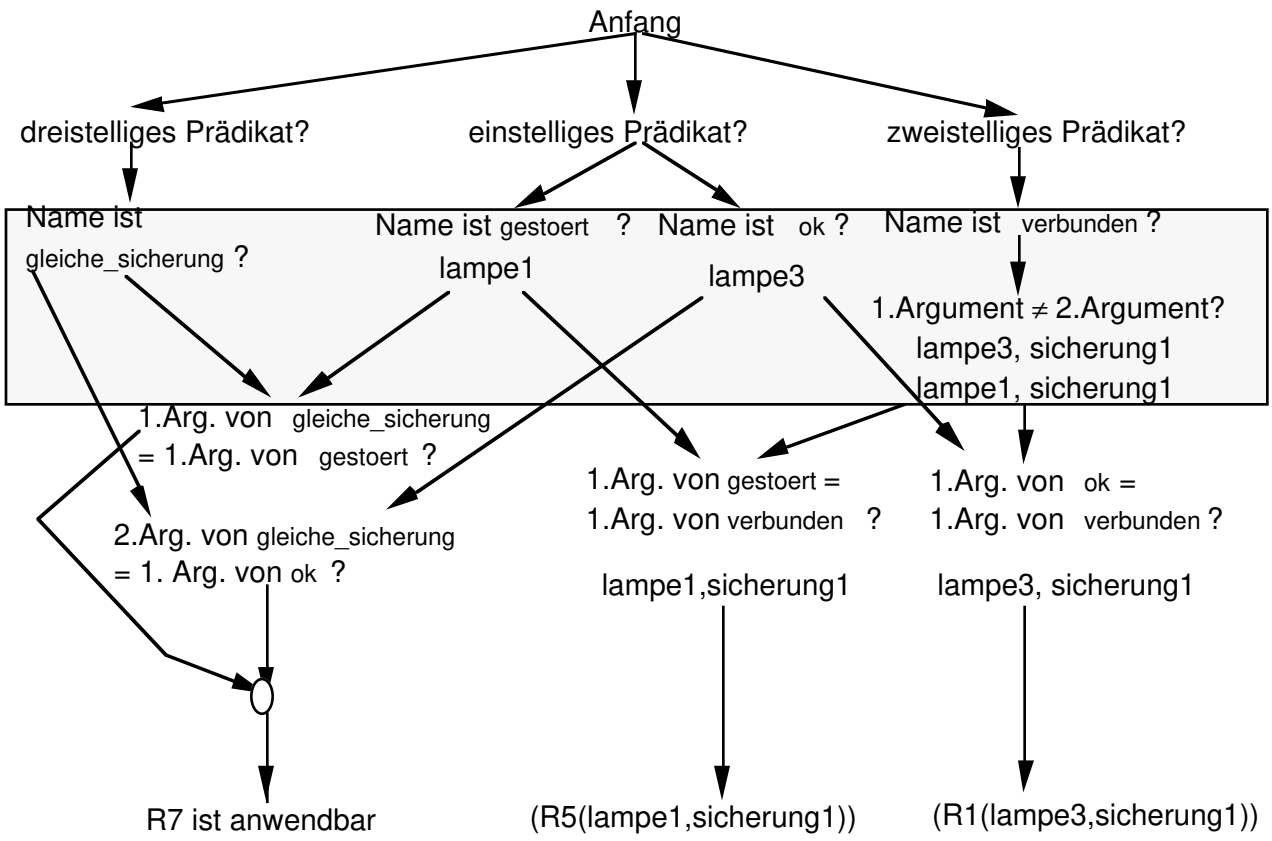
[22] Die gepunktet unterlegten Ergebnisse der Tests von Einerknoten heißen oft auch alpha-memories. Die Ergebnisse von Verbindungsknoten heißen dann beta-memories.

die entsprechende Marke aus dem Zwischenspeicher gelöscht. Die terminalen Knoten bekommen nur Marken, die alle Tests bestanden haben. Marken mit positivem Vorzeichen führen dazu, daß die Liste der abgeglichenen Kontextinhalte in der Konfliktmenge erweitert wird. Marken mit negativem Vorzeichen führen dazu, daß die Liste der abgeglichenen Kontextinhalte in der Konfliktmenge verkleinert wird. Das Netzwerk gibt dann die aktuelle, veränderte Konfliktmenge aus.

Nehmen wir an, die folgenden Kontextinhalte wären bereits von einem Rete-Netzwerk verarbeitet worden:

```
ok(lampe3). verbunden(lampe3,sicherung1).

gestoert(lampe1). verbunden(lampe1,sicherung1)
```

Dann sieht das Rete-Netz mit den Zwischenspeichern so aus:



Wenn nun die neue Marke hinzukommt

```
+dreistellig,gleiche_sicherung,1.Arg=lampe1,2.Arg=lampe3,3.Arg=sicherung1
```

so wird sie drei Mal kopiert und an alle Nachfolger des Anfangsknotens geschickt. Da es ein dreistelliges Prädikat ist, wird die Marke nur vom linkesten Einerknoten weitergeleitet. Der nachfolgende Verbindungsknoten, der bisher nichts weitergeben konnte, kann jetzt den Test vornehmen, ob das 1. Argument mit seinem rechten Zwischenspeicher (der ja lampe1 enthält) übereinstimmt. Da dies der Fall ist, reicht er die Marke weiter. Der andere nachfolgende Verbindungknoten prüft, ob sein rechter Speicherinhalt (lampe3) mit dem neuen linken übereinstimmt. Da auch das der Fall ist, werden alle Argumente für die Konfliktmenge zur Verfügung gestellt. Der terminale Knoten wird zu

```
(R7(lampe1,lampe3,sicherung1)).
```

Dies ist ein neuer Eintrag in die Konfliktmenge. Die Effizienz des Rete-Verfahrens kommt daher, daß die irrelevanten Pfade im Netz gar nicht erst geprüft werden (z.B. kein ein- oder zweistelliges Prädikat). Bei positiven Marken wird die Konfliktmenge erweitert, ohne daß bereits geprüfte Eigenschaften noch einmal betrachtet würden. Bei negativen Marken müssen alle Zwischenspeicher, die betroffen sind, neu erstellt werden. Dazu müssen auch die bereits vorgenommenen Vergleiche bei Verbindungsknoten erneut vorgenommen werden. Würde man zum Beispiel den Kontextinhalt

```
ok(lampe3)
```

wieder löschen, so müssen zwei Verbindungsknoten erneut prüfen und dann den Zwischenspeicher anpassen. In diesem Falle führt das Löschen dazu, daß die Verbindungsknoten eine Marke mit negativem Vorzeichen an die terminalen Knoten (für R7 und für R1) weiterreichen. Die Liste der abgeglichenen Kontextinhalte wird um

```
(lampe1, lampe3, sicherung1) bzw. (lampe3,sicherung1)
```

verkürzt. Damit gibt es keine abgeglichenen Kontextinhalte mehr bei diesen Regeln und sie werden ganz aus der Konfliktmenge gelöscht. Dieses Propagieren negativer Marken ist also nicht so effizient, weil Vergleiche noch einmal vorgenommen werden müssen. Bei einem Vorgehen wie in unserem Beispiel oben, in dem dauernd aus dem Kontext gelöscht wird, ist das Rete-Verfahren nicht so effizient. Neuere Verfahren optimieren gerade die Behandlung von negativen Marken und die Zwischenspeicherung bei Verbindungsknoten (Miranker 1990).

3.4.3 Erweiterung von Produktionensystemen durch Statistik

Bei ungenauem Wissen kann oft nur die Evidenz für oder gegen eine Annahme gesammelt werden. Bestimmte Aussagen erhöhen oder senken die Evidenz für eine Hypothese, sie determinieren sie nicht. Aus diesem Grund führte Shortliffe Evidenzwerte bei Aussagen ein. Ein Evidenzwert ist eine Zahl aus dem Intervall zwischen 0 und 1. Diese Evidenzwerte müssen dann verrechnet werden: einmal innerhalb einer Regel und zum anderen bei verschiedenen Ableitungen derselben Aussage. Innerhalb einer Regel kann zum Beispiel der höchste Evidenzwert einer Bedingung an den Handlungsteil weitergegeben werden, der niedrigste, oder ein Mittelwert. Wenn nun verschiedene Regeln dasselbe Ergebnis ableiten (Mehrfachableitung) nahm Shortliffe eine Evidenzverstärkung an, die er aus den Evidenzwerten der jeweiligen Handlungsteile errechnete. Bei n Ableitungen derselben Aussage mit xi als dem Evidenzwert der i-ten Ableitung der Aussage berechnet die folgende Formel den Ergebnis-Evidenzwert:

$$1-\prod_{i=1}^{n}(1-x_i)$$

In dem System MYCIN zur medizinischen Diagnose wurden Evidenzwerte verwendet. Wenn z.B. drei Mal abgeleitet wurde, daß es sich bei einem Infekt um eine Meningitis handelt, einmal mit der Evidenz 0.2, einmal mit der Evidenz 0.3 und einmal mit der Evidenz 0.8, so ergibt sich als Evidenz für eine Meningitis:

$$1-(0.8 \cdot 0.7 \cdot 0.2) = 1- 0.112 = 0.888$$

Diese Evidenzverstärkung ist nur zulässig, wenn die verschiedenen Ableitungen voneinander unabhängig sind. Bei MYCIN-Regeln kann man das kaum garantieren: indirekt mögen sie sich auf dasselbe Wissen stützen, das an verschiedenen Stellen in die Regeln hineincodiert wurde. Mostow und Swartout (Mostow, Swartout 1986) haben darauf hingewiesen, daß die Verknüpfung von Handlungs-

teilen explizit repräsentiert werden muß, wenn auch das Zusammenwirken von Regeln unter ungenauem Wissen inspizierbar und veränderbar sein soll.

3.4.4 Literatur

Davis, R., King, J.J. (1977): An Overview of Production Systems, in: Elcock, Michie (eds): *Machine Intelligence, Vol. 8*, Chichester: Ellis Horwood

Feigenbaum, E.A., Simon, H.A. (1963): Elementary Perceiver and Memorizer - Review of Experiments, in: Hoggatt, Balderston (eds): Symposion on Simulation Models - Methodology and Applications to the Behavioral Sciences, Cincinatti: Southwestern Publishers

Forgy, C.L. (1982): Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, in: *Artificial Intelligence*, 19, 17-37

Miller, G.A. (1956): The Magical Number 7 Plus or Minus Two - Some Limits on our Capacity for Processing Information, *Psychological Review* 63, 81-97.

Miranker, D.P. (1990): TREAT: A New and Efficient Match Algorithm for AI Production Systems, London: Pitman & San Mateo: Morgan Kaufmann

Mostow, J., Swartout, B. (1986): Towards Explicit Integration of Knowledge in Expert Systems -- An Analysis of MYCIN's Therapy Selection Algorithm, in: Procs. of AAAI-86, Philadelphia, 928-935.

Post, E.(1943): Formal Reductions of the General Combinatorial Problem, in: *American Journal of Mathematics*, 65, 197-268.

Shortliffe, E. (1976): Computer-Based Medical Consultations: MYCIN, American Elsevier

Simon, Herbert (1978): Acht Vorlesungen über kognitive Psychologie, gehalten an der Universität Hamburg, Fachbereich Psychologie, 1978