# Nachfragende und konzeptbasierte Metainterpreter in Prolog

Jens Lehmann

1. September 2004

Der vorliegende Beleg entstand während der Vorlesung „Wissensentdeckung, Lernen und Schließen mit Case-based Reasoning und Neuronalen Netzen“ von Dr. Petersohn an der Fakultät Informatik der Technischen Universität Dresden im Sommersemester 2004.

Es wird erklärt, wie mit Hilfe von Prolog ein Metainterpreter geschrieben werden kann, der (zur Lösung einer Anfrage benötigtes, aber nicht in einer gegebenen Wissensbasis vorhandenes) Wissen vom Benutzer nachfragt um eine Anfrage zu lösen. Außerdem wird erklärt, wie Hierarchien von Konzepten unterstützt werden können. Für die zugrundeliegende Wissensbasis wird eine einfache Prolog-Syntax unterstützt.

# Inhaltsverzeichnis

# 1. Einleitung

Die Arbeit beschäftigt sich mit Implementierungen von Metainterpretern in Prolog. Nach einer kurzen Einführung in die Funktionsweise eines Metainterpreters wird in Abschnitt 2 auf die konkrete Implementierung eingegangen. Davon ausgehend werden stufenweise immer komplexere Metainterpreter erklärt. Abschnitt 3 beschäftigt sich mit nachfragenden Metainterpretern und Abschnitt 4 geht auf mögliche Realisierungen von Konzepthierarchien ein. Für das Verständnis des Belegs wird vorausgesetzt, dass die Funktionsweise von Prolog bekannt ist.

## 1.1. Was ist ein Metainterpreter?

Ein Metainterpreter in Prolog ist ein selbst in Prolog geschriebenes Programm, welches eine Sprache interpretiert, die wiederum eine Teilmenge von Prolog ist. Metainterpreter können eingesetzt werden um die Funktionsweise des normalen Prologinterpreters zu ändern, um mehr Informationen (z.B. die Anzahl der Inferenzschritte) auszugeben oder um die Sprache um neue Elemente zu erweitern. Insbesondere der letzte Aspekt wird in dieser Arbeit behandelt. Prolog wird in der Richtung erweitert, dass ein nicht vorhandener Fakt nicht als „falsch“ interpretiert wird, sondern der Benutzer in dem Fall selber entscheiden kann, ob der Fakt wahr oder falsch ist. Zum anderen sollen Konzepthierarchien untersützt werden, was in Abschnitt 4 erklärt wird.

## 1.2. Wie kann man in Prolog einen Metainterpreter schreiben?

Der einfachste bzw. der triviale Metainterpreter in Prolog sieht folgendermaßen aus:

**Quelltext 1.1 (Direktlösungsregel)**

```
agent(Goal) :- Goal.
```

Der Interpreter ist in diesem Fall nur das Prädikat `agent/1`, welches eine Anfrage entgegennimmt und diese Anfrage dann vom Prolog-Interpreter selbst lösen lässt. Alle Anfragen an den Metainterpreter müssen also die Form `agent(X)` haben, wobei `X` die Anfrage an den Metainterpreter ist. Natürlich ist dieser Metainterpreter, sofern er nur aus dieser Klausel besteht, nicht sinnvoll, da er gegenüber dem internen Prologinterpreter keine Vorteile hat und nur einen zusätzlichen Inferenzschritt bedeutet. In den nächsten Abschnitten werden schrittweise praktisch einsetzbare Metainterpreter konstruiert.

# 2. Implementierung eines Standard-Metainterpreters

Zur Veranschaulichung der Funktionsweise des in diesem Abschnitt vorgestellten Interpreters wird eine kleine Wissensbasis verwendet:

**Quelltext 2.1 (Wissensbasis zu Elefantenarten)**
```
concept(T, afrikanischer_elefant) :-
  ort(T, afrika),
  stosszahnrichtung_oben(T).

concept(T, waldelefant) :-
  ort(T, afrika),
  not(stosszahnrichtung_oben(T)).

concept(T, asiatischer_elefant) :-
  ort(T, asien),
  (stosszahn_nicht_sichtbar(T);
  stosszahnrichtung_oben(T)).
```

Die Wissensbasis enthält eine extrem vereinfachte Klassifizierung der drei Elefantenarten. Der afrikanische Elefant und der Waldelefant leben in Afrika, wobei sie sich (unter anderem) dadurch unterscheiden, dass beim Waldelefanten die Stosszähne Richtung Boden zeigen, während das beim afrikanischen Elefanten nicht der Fall ist. Beim asiatischen Elefant gilt als zusätzliches Merkmal, dass einige weibliche Tiere keine äußerlich sichtbaren Stoßzähne haben.

Die im Quelltext 2.1 gezeigte Wissensbasis ist ein einfacher, aber dennoch typischer Fall für die Unterscheidung nach Konzepten. Es soll jetzt ein Metainterpreter konstruiert werden, der in der Lage ist Anfragen an diese Wissensbasis zu lösen. Dazu ist es nicht notwendig, dass der Metainterpreter die volle Prolog-Syntax interpretieren kann. Die Syntax der Wissensbasen soll auf Klauseln, in deren Klauselkörper Disjunktionen, Konjunktionen und Negationen vorkommen können, beschränkt werden. Da eine Anfrage schrittweise zerlegt wird, benötigt unser Metainterpreter für alle auftretenden Fälle jeweils Regeln, wie er vorgehen soll. Diese Regeln werden im Folgenden erklärt. (Zur einfachereren Erklärung wurde jeder Regel ein Name zugeordnet.)

**Quelltext 2.2 (Negationsregel)**
```
solve(not(Goal)):-
  !,
  not(solve(Goal)).
```

Falls die Anfrage die Form `not(Goal)` hat, dann soll versucht werden das Ziel `Goal` zu lösen. Falls das gelingt, so ist die gesamte Anfrage nicht erfolgreich. Falls das nicht gelingt, so ist die gesamte Anfrage erfolgreich. Das ist das Prinzip von Negation als Fehlschlag. Interessant ist beim Schreiben eines Metainterpreters das korrekte Setzen von Cuts. Es ist bekannt, dass, wenn die Anfrage die Form `not(Goal)` hat, nur die Negationsregel sinnvoll angewendet werden kann. Deswegen kann hier ein Cut gesetzt werden, der im Ableitungsbaum von Prolog die weiteren Möglichkeiten abschneidet. Der Cut wird bereits

vor `not(solve(Goal))` gesetzt, da der Ableitungsbaum unabhängig davon, ob die gesamte Anfrage erfolgreich ist oder fehlschlägt, abgeschnitten werden kann. (Ein Cut selbst ist immer erfolgreich.)

**Quelltext 2.3 (Konjunktionsregel)**
```
solve((Goal1,Goal2)):-
  !,
  solve(Goal1),
  solve(Goal2).
```

Die Konjunktionsregel ist ähnlich aufgebaut und zeigt wie die Anfrage schrittweise zerlegt wird. Man beachte die Klammerung im Klauselkopf. Der Cut kann analog zur Negationsregel wieder am Anfang gesetzt werden.

**Quelltext 2.4 (Disjunktionsregel)**
```
solve((Goal1;_)) :- solve(Goal1), !.
solve((_;Goal2)) :- !, solve(Goal2).
```

Die Disjunktionsregel ist weniger offensichtlich. Man könnte vermuten, dass es einfacher ist die prologinterne Disjunktion zu verwenden („`;`“-Operator). Das führt jedoch zu Problemen, da die Disjunktion in Prolog folgendermaßen definiert wird:

**Quelltext 2.5 (Definition der Disjunktion in Prolog)**
```
Goal1 ; _Goal2 :- Goal1.
_Goal1 ; Goal2 :- Goal2.
```

Daraus folgt, dass selbst wenn die erste Klausel erfolgreich ist (d.h. `Goal1` ist wahr), mit der zweiten Klausel ein weiterer Backtracing-Punkt existiert. Dass soll verhindert werden, da dies später bei der Implementierung eines Metainterpreters, der dem Benutzer bei nicht vorhandenem Wissen Fragen stellen soll, dazu führen kann, dass unnötige Fragen gestellt werden, d.h. es könnte vorkommen, dass selbst bei einem erfolgreichen `Goal1` noch nach `Goal2` (bzw. einem Teilziel davon) gefragt wird.

Die Disjunktionsregel ist also so implementiert, dass erst versucht wird das erste Teilziel `Goal1` zu lösen. Falls das erfolgreich ist, wird der Cut ausgeführt und somit die zweite Klausel und alle weiteren nicht beachtet. Falls `Goal1` jedoch fehlschlägt, dann wird die zweite Klausel ausgeführt. Dort wird in jedem Fall ein Cut ausgeführt, damit sichergestellt ist, dass die folgenden `solve`-Klauseln im Metainterpreter nicht mit einer Disjunktion von Formeln arbeiten müssen.

**Quelltext 2.6 (Boolesche Regel)**
```
solve(true) :- !.
solve(fail) :- !, fail.
```

Die Boolesche Regel wertet das Schlüsselwort `true` als wahr aus und schneidet den Lösungsbaum ab. Das Schlüsselwort `fail` wird als Fehlschlag interpretiert. Die Booleschen Regeln sind notwendig, damit Fakten der Wissensbank ausgewertet werden können.

**Quelltext 2.7 (Beispiel für „Fakten“ im Sinne des Metainterpreters)**
```
ort(elefant, afrika) :- fail.
ort(elefant, asien).
```

Insbesondere die erste Klausel ist interessant, da das die Kodierung dafür ist, dass ein bestimmter Fakt nicht wahr ist. Beim nachfragenden Metainterpreter, auf den hingearbeitet wird, gibt es also praktisch drei Zustände für einen Fakt: wahr, falsch und unbestimmt.

**Quelltext 2.8 (Klauselkörperregel)**
```
solve(Goal):-
  clause(Goal,Body),
  solve(Body).
solve(Goal):-
  clause(Goal, _), !, fail.
```

Die Klauselkörperregel verwendet das in Prolog bereits vordefinierte Prädikat `clause/2`. `clause(Goal, Body)` sucht Klauseln im Programm, deren Klauselköpfe mit `Goal` und Klauselkörper mit `Body` unifizierbar sind. Falls so eine Klausel gefunden wird, dann wird `solve` auf dem Klauselkörper aufgerufen. Das ist genau das Prinzip nach dem der interne Prologinterpreter funktioniert. Falls es sich bei `Goal` um einen Fakt handelt, dann wird `Body` mit dem Atom `true` unifiziert. Aus diesem Grund ist die erste Klausel der Booleschen Regel wichtig.

Die `clause`-Anweisung gibt beim Backtracing natürlich schrittweise alle unifizierbaren Klauseln zurück. Da eventuell nachfolgende `solve`-Klauseln nicht mit Klauselköpfen arbeiten sollen, wird die zweite Klausel in der Klauselkörperregel eingebaut. Sie hat eine ähnliche Funktion wie die Cuts in den Regeln zuvor. Falls `clause` ein Ergebnis zurückgibt, so wird der Ableitungsbaum abgeschnitten und die Auswertung schlägt fehl.

Durch die Negations-, Konjunktions-, Disjunktions- und Klauselkörperregel wurde erreicht, dass die Anfrage komplett syntaktisch zerlegt wurde. Die Auswertung eines nicht weiter zerlegbaren (atomaren) Teilziels erfolgt dann mit der Booleschen Regel. An einem konkreten Beispiel soll jetzt die Funktionsweise demonstriert werden:

**Quelltext 2.9 (Beispiel für vorhandenes Wissen)**
```
ort(ele1, afrika).
stosszahnrichtung_oben(ele1).
```

Über einen Elefant ist also bekannt, dass er in Afrika lebt und die Stosszahnspitzen nach oben zeigen. Laut unserer Wissensbasis 2.1 handelt es sich also um einen afrikanischen Elefanten. Das wollen wir mit Hilfe des Metainterpreters überprüfen. Wir stellen eine Anfrage `solve(concept(ele1, X))`. Die erste geeignete Regel ist die Klauselkörperregel. `concept(ele1, X)` lässt sich mit dem Klauselkopf `concept(T, afrikanischer_elefant)` in unserer Wissensbasis unifizieren. Demzufolge wird jetzt `solve` für den Klauselkörper `(ort(ele1, afrika), stosszahnrichtung_oben(ele1))` aufgerufen. Darauf

lässt sich die Konjunktionsregel anwenden, wobei zuerst versucht wird das erste Teilziel `ort(ele1, afrika)` zu lösen. Dies geschieht mit Hilfe der Klauselkörperregel, da sich das Ziel mit dem vorhandenen Fakt `ort(ele1, afrika)` unifizieren lässt. Für Fakten wird `Body` in der Klauselkörperregel `true`. `true` lässt sich erfolgreich mit der Booleschen Regel lösen. Damit ist die Klauselkörperregel für `ort(ele1, afrika)` und somit das erste Teilziel in der Konjunktionsregel erfolgreich. Das zweite Teilziel `stosszahnrichtung_oben(ele1)` lässt sich analog mit der Klauselkörperregel und anschließend der Booleschen Regel erfolgreich lösen. Demzufolge ist die angewandte Konjunktionsregel erfolgreich und damit auch die allererste Klauselkörperregel für `concept(ele1, X)`. Die Lösung `X = afrikanischer_elefant` wird vom Prolog-System ausgegeben. Mit Hilfe des vom Metainterpreter A.2 (siehe Anhang) ausgegeben Protokolls kann man die Verarbeitung von Anfragen nachvollziehen. Der Quellcode des vorgestellten Metainterpreters ist in A.1 noch einmal zusammenhängend aufgelistet.

# 3. Implementierung eines nachfragenden Metainterpreters

## 3.1. Funktionsprinzip

Mit dem vorgestellten Metainterpreter wurde eine Basis zur Interpretation einer vereinfachten Prolog-Syntax geschaffen. Prinzipiell wurde damit noch kein Mehrwert gegenüber dem prologinternen Interpreter erreicht, da jede vom Metainterpreter ausgewertete Anfrage auch vom prologinternen Interpreter ausgewertet werden kann. Der Metainterpreter kann jedoch erweitert werden um das Nachfragen von nicht vorhandenem Wissen zu erlauben. Normalerweise wird in Prolog eine Anfrage nach einem nicht vorhandenen Fakt mit „No“ beantwortet. Es soll jetzt ein Interpreter geschrieben werden, der in diesem Fall den Benutzer fragt, ob der Fakt wahr ist oder nicht. Außerdem sollen die Antworten auf Fragen gespeichert werden, so dass der Benutzer innerhalb einer Anfrage nicht mehrfach die gleiche Frage beantworten muss.

Um diese Ziele zu erreichen, werden zwei neue Regeln eingeführt. Eine Regel, die dem Benutzer Fragen stellt, falls dies notwendig ist, und eine Regel, die überprüft, ob zu einer Anfrage bereits eine entsprechende Antwort gegeben wurde.

**Quelltext 3.1 (Frageregel)**
```
solve(Goal):-
  question(Goal,Answer),
  addanswer(Goal,Answer).
```

Die Frageregel stellt dem Benutzer die Frage, ob ein Ziel wahr ist. Die Antwort wird eingelesen und zu einer Antwortdatenbank hinzugefügt.

**Quelltext 3.2 (Frage stellen)**
```
question(Goal,Answer):-
  writef('Is %q true? (y/n) ', [Goal]),
  get_single_char(Answer),
  (Answer = 121; Answer = 110) ->
  (
    ((Answer = 121) -> print('yes'); print('no')),
    nl
  );
  write('Illegal answer, enter y for yes and n for no.'),
  nl, question(Goal, Answer).
```

Beim Stellen der Frage sind „y“ (Code 121) und „n“ (Code 110) als Antworten erlaubt. Die Frage wird so lange gestellt, bis eine dieser beiden Antworten erfolgt (es werden if-then-else-Bedingungen verwendet). Am Ende der Ausführung ist in `Answer` die erhaltene Antwort gespeichert.

**Quelltext 3.3 (Antwort speichern)**
```
addanswer(Goal, 121) :- assert(answer(Goal)).
addanswer(Goal, 110) :- assert(answer(Goal):-fail), fail.
```

Nachdem eine Frage gestellt wurde, wird die Antwort gespeichert. Das geschieht mit dem Prädikat `assert/1`, welches einen Fakt bzw. eine Klausel als Argument entgegennimmt und diese dynamisch dem Prologprogramm hinzufügt. Die erste Klausel speichert demzufolge ab, dass `Goal` mit ja, und die zweite Klausel speichert, dass `Goal` mit nein beantwortet wurde. Das Prädikat `answer/1` kann später genutzt werden um abzufragen, ob zu einer Frage schon eine Antwort existiert. Man beachte auch, dass die erste Klausel immer erfolgreich ist, während die zweite Klausel immer fehlschlägt. Es hängt also von der gegebenen Antwort ab, ob die Frageregel erfolgreich ist oder nicht.

**Quelltext 3.4 (Antwortregel)**

```
solve(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).
```

Die Antwortregel kontrolliert, ob ein gegebenes Ziel bereits beantwortet wurde. Falls eine Antwort gefunden wird, so wird - unabhängig davon wie die Antwort lautet - der Lösungsbaum abgeschnitten, da bei einer mit „n“ beantworteten Frage der Metainterpreter natürlich nicht noch weiter versuchen soll das Ziel doch noch zu erfüllen. Die letzte Anweisung `answer(Goal)` schlägt dann fehl, falls die Klausel `answer(Goal):-fail` existiert, und ist erfolgreich, falls `answer(Goal)` als Fakt existiert (siehe `assert`-Anweisungen). Natürlich muss die Antwortregel im Quellcode vor der Frageregel stehen, da vor einer möglichen Frage erst nachgeschaut wird, ob schon eine Antwort existiert.

**Quelltext 3.5**

```
agent(Goal) :- retractall(answer(_)), solve(Goal).
```

Damit bei jeder neuen Anfrage die bisher gegebenen Antworten gelöscht werden (normalerweise werden diese für die Dauer einer Prolog-Sitzung gespeichert) wird in der Startfunktion `agent/1` das Prädikat `retractall/1` aufgerufen, welches alle Klauseln mit der angegeben Form entfernt.

*Anmerkung:* Die verwendete Klauselkörperregel (2.8) ist in dem nachfragenden Metainterpreter eine Einschränkung. Falls alle Klauseln zu einem Klauselkopf fehlschlagen, so trifft das in unserem Sinn noch keine Aussage darüber, ob der Klauselkopf wahr oder falsch ist. Man müsste demzufolge den Benutzer fragen, ob der Klauselkopf wahr ist. Vom praktischen Standpunkt her ist es jedoch günstiger die Vereinbarung zu treffen, dass ein Klauselkopf fehlschlägt, falls alle Klauselkörper mit diesem Klauselkopf fehlschlagen. Das ist in den meisten Fällen das gewünschte Systemverhalten. Hätte man als Wissensbasis nur die Klausel `t1(test) :- t2(test).` und stellt die Anfrage, ob `t1(test)` wahr ist, so wird erst gefragt, ob `t2(test)` wahr ist, und falls das fehlschlägt (Antwort „n“), so schlägt auch `t1(test)` fehl.

Mit den vorgestellten Erweiterungen ist der Metainterpreter in unserem konkreten Fall in der Lage interaktiv die Art eines Elefanten zu bestimmen, über den bis jetzt noch keine Daten gespeichert wurden. Der Metainterpreter nimmt dem Benutzer also die Entscheidung ab zu bestimmen, welche Daten benötigt werden. Eine Anfrage könnte zum Beispiel folgende Gestalt haben:

**Beispiel 3.6**

```
?- agent(concept(ele, Art)).
Is ort(ele, afrika) true? (y/n) yes
Is stosszahnrichtung_oben(ele) true? (y/n) no

Art = waldelefant

Yes
```

## 3.2. Verbesserung der Effizienz

Der vorgestellte Metainterpreter arbeitet korrekt. Es ist jedoch möglich die Effizienz zu verbessern. Unter Effizienz ist dabei die Anzahl der Fragen und nicht die Anzahl der Inferenzschritte zu verstehen. Man nehme an, dass folgende Wissensbasis gegeben ist:

**Quelltext 3.7 (Schwächen des Metainterpreters)**

```
concept(D,a) :- t1(D), t2(D).
concept(D,b) :- t3(d); t1(D).
```

Bei der Anfrage `concept(t,X)` würde der Metainterpreter erst versuchen die erste Klausel zu erfüllen, also nach `t1(t)` fragen. Antwortet man mit „y“, dann wird nach `t2(t)` gefragt. Bei Antwort „n“ schlägt die erste Klausel fehl und durch das Backtracing in der Klauselkörperregel springt der Metainterpreter zur zweiten Klausel. Dort würde er zuerst nach `t3(t)` fragen. Diese Frage ist jedoch überflüssig, da die rechte Seite der Disjunktion bereits positiv beantwortet werden kann. Analoge Szenarien lassen sich auch für die Konjunktion finden.

Ziel ist es jetzt also einen Metainterpreter zu konstruieren, der bei einer Konjunktion oder Disjunktion von Teilzielen versucht diese zu lösen ohne Fragen zu stellen. Um so einen Metainterpreter zu konstruieren, benötigt man die dreiwertige Logik. Da das System nicht voraussagen kann, welche Antworten der Benutzer auf Fragen gibt, kann ein gegebenes Ziel drei mögliche Wahrheitswerte haben: „richtig“, „falsch“ und „möglich“ (bzw. „unbestimmt“ oder „richtig oder falsch“). Diese Werte können mit 0 (richtig), 1 (falsch) und 2 (richtig oder falsch) kodiert werden. Um das Vorhaben zu lösen, kann man zuerst Tabellen für die Disjunktion bzw. Konjunktion in der dreiwertigen Logik aufstellen. Das wurde in den Tabellen 1 und 2 gemacht.

Im Gegensatz zur zweiwertigen Logik gibt es neun statt vier zu betrachtende Fälle. Die Kreuze in der letzten Spalte wurden an den Stellen gesetzt, an denen es möglich ist eine Konjunktion bzw. Disjunktion auszuwerten, ohne das der Benutzer Fragen beantworten muss, d.h. an den Stellen, an denen die Verknüpfung immer wahr (0) oder immer falsch (1) ist. Um die Erkenntnisse nutzen zu können, werden für den Metainterpreter zwei weitere Prädikate benötigt. Es wird das Prädikat `solve_sure_success/1` eingeführt, welches eine Anfrage entgegennimmt und erfolgreich ist, wenn die Anfrage erfolgreich ist ohne dem Benutzer Fragen zu stellen. Falls `solve_sure_success` fehlschlägt, dann lässt sich über die Anfrage keine Aussage treffen. Außerdem wird analog dazu

| | a | b | a;b | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | x |
| 2 | 0 | 1 | 1 | x |
| 3 | 0 | 2 | 2 | |
| 4 | 1 | 0 | 1 | x |
| 5 | 1 | 1 | 1 | x |
| 6 | 1 | 2 | 1 | x |
| 7 | 2 | 0 | 2 | |
| 8 | 2 | 1 | 1 | x |
| 9 | 2 | 2 | 2 | |

Tabelle 1: Disjunktion

| | a | b | a,b | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | x |
| 2 | 0 | 1 | 0 | x |
| 3 | 0 | 2 | 0 | x |
| 4 | 1 | 0 | 0 | x |
| 5 | 1 | 1 | 1 | x |
| 6 | 1 | 2 | 2 | |
| 7 | 2 | 0 | 0 | x |
| 8 | 2 | 1 | 2 | |
| 9 | 2 | 2 | 2 | |

Tabelle 2: Konjunktion

das Prädikat `solve_sure_fail/1` eingeführt, welches fehlschlägt, wenn die Anfrage unabhängig von den Antworten, die der Benutzer gibt, fehlschlägt. Wenn `solve_sure_fail` erfolgreich ist, dann lässt sich über die Anfrage keine Aussage treffen. Die konkrete Implementierung dieser Prädikate wird später erklärt.

Zuerst soll erläutert werden wie mit Hilfe der Wahrheitstabellen für dreiwertige Logik und den beiden Prädikaten der Metainterpreter in gewünschter Weise erweitert werden kann. Änderungen gibt es im Prädikat `agent/1`, der Konjunktions- und der Disjunktionsregel.

**Quelltext 3.8**
```
agent(Goal) :- retractall(answer(_)), presolve(Goal).
presolve(Goal) :- not(solve_sure_fail(Goal)), !, fail.
presolve(Goal) :- solve(Goal).
```

Im Prädikat `agent/1` wird das neue Prädikat `solve_sure_fail/1` genutzt um zu testen, ob das Ziel überhaupt erfüllbar ist. Würde man an das folgende Programm eine Anfrage `agent(concept(test,X))` stellen, so würde man eine Anfrage sparen:

**Quelltext 3.9**
```
concept(D,a) :- not(t0(D)).
t0(D) :- t1(D).
t0(test).
```

Man könnte in `presolve` auch testen, ob es eine Lösung für das Programm ohne Nachfragen gibt und diese sofort ausgeben:

**Quelltext 3.10**
```
presolve(Goal) :- solve_sure_success(Goal); solve(Goal).
```

Der Nachteil dieser Variante ist, dass Lösungen doppelt ausgegeben werden, da jede Lösung von `solve_sure_success` auch eine Lösung von `solve` ist. Aus diesem Grund wurde die Variante in Quelltext 3.8 gewählt.

Die entscheidenden Änderungen gibt es in der Konjunktions- und der Disjunktionsregel, die im folgenden deshalb als erweiterte Konjunktions- bzw. erweiterte Disjunktionsregel bezeichnet werden.

**Quelltext 3.11 (erweiterte Konjunktionsregel)**

```
solve((Goal1,_)) :-
  not(solve_sure_fail(Goal1)),
  !, fail.
solve((_,Goal2)) :-
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1,Goal2)) :-
  solve_sure_success(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  !,
  solve(Goal1),
  solve(Goal2).
```

Betrachten wir zuerst die ersten drei Klauseln der erweiterten Konjunktionsregel. Diese Klauseln stellen die direkte Abbildung von Tabelle 2 in den Metainterpreter dar. Die erste Klausel besagt, dass wenn das erste Teilziel falsch ist (Kodierung 0), die Disjunktion beider Teilziele auch falsch ist. Das entspricht den Kreuzen an Position 1, 2 und 3 in der Tabelle. Analog deckt die zweite Klausel die Kreuze an Position 1, 4 und 7 ab (praktisch wird Position 1 nicht auftreten, da dies bereits von der ersten Klausel abgefangen wird). Die dritte Klausel entspricht letztendlich dem Kreuz an Position 5 in der Tabelle.

Wenn die drei ersten Klauseln fehlschlagen, so muss versucht werden ein Teilziel durch Nachfragen zu lösen (`solve(Goal1)`). Zu beachten ist dabei, dass durch das Nachfragen neue Informationen gewonnen werden können. Es könnte deshalb passieren, dass das zweite Teilziel mit Hilfe der neuen Informationen auch ohne Nachfragen gelöst werden kann. Das wird durch die Klauseln 4 (zweites Teilziel ist sicher erfolgreich) und 5 (zweites Teilziel schlägt sicher fehl) erledigt. Klausel 6 ist die ursprünglich vorhandene Regel, bei der auch zur Lösung des zweiten Teilziels Fragen an den Benutzer gestellt werden.

**Quelltext 3.12 (erweiterte Disjunktionsregel)**

```
solve((Goal1;_)) :-
  solve_sure_success(Goal1),
  !.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
```

```
solve((Goal1;Goal2)) :-
  not(solve_sure_fail(Goal1)),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1;_)) :-
  solve(Goal1),
  !.
solve((_;Goal2)) :-
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
solve((_;Goal2)) :-
  !,
  solve(Goal2).
```

Die erweiterte Disjunktionsregel hat den gleichen Aufbau wie die erweiterte Konjunktionsregel. Klausel 1 deckt die Positionen 4, 5 und 6 in Tabelle 1 ab. Bei Klausel 2 sind es die Positionen 2, 5 und 7 und bei Klausel 3 die Position 1. Klausel 4 ist die erste Klausel der ursprünglichen Disjunktionsregel und besagt, dass die Anfrage erfolgreich ist, wenn das erste Teilziel gelöst werden kann. Durch die neu gewonnene Information im Lösungsprozess ist es analog zur erweiterten Konjunktionsregel möglich, dass das zweite Teilziel ohne Fragen gelöst werden kann (Klauseln 5 und 6). Klausel 7 ist letztendlich die zweite Klausel der ursprünglichen Disjunktionsregel und besagt, dass abhängig davon, ob das zweite Teilziel erfolgreich gelöst werden kann, die Anfrage erfolgreich ist.

Durch die Konstruktion der erweiterten Konjunktions- und Disjunktionsregel über Wahrheitstabellen der dreiwertigen Logik haben sich relativ komplexe Regeln ergeben. Ein anfangs einleuchtender Ansatz wäre es gewesen im Falle der Konjunktion nur die Klauseln 2 und 6 zu verwenden. Ausformuliert würde das bedeuten: Schlägt in einer Konjunktion das zweite Teilziel fehl, so schlägt die Anfrage fehl, ansonsten wird versucht nacheinander das erste und zweite Teilziel zu lösen. Anhand von einigen Beispielen soll gezeigt werden, dass die zusätzliche Komplexität der erweiterten Regeln tatsächlich sinnvoll ist.

**Quelltext 3.13**

```
concept(D,a) :- t0(D),t1(D).
t1(D) :- t2(D).
t1(D) :- t0(D).
```

Das obige Beispiel zeigt, dass Klausel 5 benötigt wird um `concept(test,X)` mit nur einer Frage zu lösen. Zuerst wird `t0(test)` vom Benutzer abgefragt. Falls der Nutzer mit „y“ antwortet, dann ist dadurch Information gewonnen worden, die durch Klausel 5 jetzt verwendet werden kann um das zweite Teilziel `t1(test)` erfolgreich zu beantworten. Ohne diese Klausel würde der Metainterpreter versuchen `t1(test)` mit der Klauselkörperregel zu lösen, d.h. er würde nach `t2(test)` fragen um die erste Klausel mit Klauselkopf `t1(D)` zu erfüllen.

Das Ergebnis bleibt natürlich das gleiche, unabhängig davon, wie die Frage beantwortet wird. (Entweder die Antwort ist „y“ oder es wird zur nächsten Klausel gesprungen, die dann erfolgreich ist.)

Ähnlich kann man auch zeigen, dass der einfache Ansatz bei der Disjunktion nur die sofort einleuchtenden Regeln 2, 4 und 7 zu verwenden nicht optimal ist.

**Quelltext 3.14**

```
concept(D,a) :- t1(D);fail.
t1(D) :- t2(D).
t1(D) :- t3(D).
t3(test).
```

Um mit dieser Wissensbasis die Anfrage `concept(test,X)` ohne eine Frage zu lösen, wird Klausel 1 der erweiterten Disjunktionsregel benötigt. Klausel 1 prüft, ob das erste Teilziel immer wahr ist, was hier der Fall ist. Ohne diese Regel würde der Metainterpreter versuchen `t1(test)` mit der Klauselkörperregel zu lösen, was zu einer überflüssigen Frage nach `t2(test)` führt.

Diese beiden Beispiele sollen zur Veranschaulichung der Funktionsweise genügen. Man könnte für jede Klausel der beiden Regeln Beispiele konstruieren. Wichtig ist, dass die Erweiterung des Metainterpreters nicht das Ergebnis verändert, sondern nur in vielen Fällen die Anzahl der gestellten Fragen verringert.

Was noch fehlt um den erweiterten Metainterpreter zu vervollständigen, ist die Implementierung von `solve_sure_fail` und `solve_sure_success`. Beide Prädikate sind selbst wie ein Metainterpreter aufgebaut. Es fehlt natürlich in beiden Fällen die Frageregel, da vom Benutzer keine neuen Informationen abgefragt werden sollen. Zunächst wird `solve_sure_fail` betrachtet.

**Quelltext 3.15 (solve_sure_fail)**

```
solve_sure_fail(not(Goal)):-
  !,
  not(solve_sure_success(Goal)).

solve_sure_fail((Goal1,Goal2)):-
  !,
  solve_sure_fail(Goal1),
  solve_sure_fail(Goal2).

solve_sure_fail((Goal1;_)) :- solve_sure_fail(Goal1), !.
solve_sure_fail((_;Goal2)) :- !, solve_sure_fail(Goal2).

solve_sure_fail(true) :- !.
solve_sure_fail(fail) :- !, fail.

solve_sure_fail(Goal):-
  clause(Goal,Body),
  solve_sure_fail(Body).
solve_sure_fail(Goal):-
```

```
  clause(Goal, _), !, fail.

solve_sure_fail(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

solve_sure_fail(_).
```

Geändert wurde hier gegenüber dem Metainterpreter aus 3.1 die Negationsregel (die Namen der Regeln wurden im Quelltext der Kompaktheit halber weggelassen). Wenn das Ziel `Goal` immer wahr ist, dann schlägt die Negationsregel fehl. Das ist korrekt, da vereinbart wurde, dass `solve_sure_fail` dann fehlschlagen soll, wenn eine Anfrage keine Lösung hat. Die beiden Prädikate `solve_sure_fail` und `solve_sure_success` hängen also voneinander ab. Eine weitere Änderung ist die letzte Klausel, die für jedes beliebige Ziel erfolgreich ist. Der Hintergrund ist, dass diese Klausel nur erreicht wird, falls die Anfrage nicht bereits durch die Boolesche Regel oder die Antwortregel negativ beantwortet wurde. In diesem Fall würde im nachfragenden Metainterpreter eine Frage gestellt werden. Da nicht vorausgesagt werden kann, wie diese Frage beantwortet wird, wird über diese Klausel jedes Ziel in `solve_sure_fail` erfolgreich. Nach der Spezifikation von `solve_sure_fail` bedeutet das, dass keine Aussage darüber getroffen werden kann, ob das Ziel wahr oder falsch ist.

**Quelltext 3.16 (solve_sure_success)**

```
solve_sure_success(not(Goal)):-
  !,
  not(solve_sure_fail(Goal)).

solve_sure_success((Goal1,Goal2)):-
  !,
  solve_sure_success(Goal1),
  solve_sure_success(Goal2).

solve_sure_success((Goal1;_)) :- solve_sure_success(Goal1), !.
solve_sure_success((_;Goal2)) :- !, solve_sure_success(Goal2).

solve_sure_success(true) :- !.
solve_sure_success(fail) :- !, fail.

solve_sure_success(Goal):-
  clause(Goal,Body),
  solve_sure_success(Body).
solve_sure_success(Goal):-
  clause(Goal, _), !, fail.

solve_sure_success(Goal):-
  clause(answer(Goal),_),
```

```
!,
answer(Goal).
```

Analog zu `solve_sure_fail` ist `solve_sure_success` aufgebaut. Die Unterschiede bestehen darin, dass in der Negationsregel `solve_sure_fail` aufgerufen wird und die letzte Klausel fehlt. Man könnte um die Analogie zu veranschaulichen als letzte Klausel `solve_sure_success(_) :- fail.` einfügen, d.h. eine Klausel, die für jede Anfrage fehlschlägt. Das ist jedoch nicht notwendig, da im prologinternen Interpreter eine Anfrage, für die es keine Lösung gibt, ohnehin fehlschlägt. Der Fehlschlag bedeutet, dass sich über die Anfrage `Goal` keine Aussage treffen lässt.

Damit ist der erweiterte nachfragende Metainterpreter vollständig beschrieben. Es sind noch weitere Verbesserungen am Interpreter möglich, von denen einige hier kurz beschrieben werden sollen. Man könnte zum Beispiel noch die Klauselkörperregel so modifizieren, dass nach jedem Backtracing (d.h. wenn von einer möglichen Klausel zur nächsten gesprungen wird) kontrolliert wird, ob es ein Ziel gibt, was durch die neu hinzugewonnenen Informationen jetzt ohne Frage gelöst werden kann bzw. ob ohne Frage gezeigt werden kann, dass die Anfrage nicht erfüllt werden kann.

Außerdem erkennt der Metainterpreter natürlich keine Zusammenhänge zwischen Teilzielen. Hätte man als Wissensbasis nur die Klausel `concept(a) :- t,not(t).` und stellt die Anfrage `agent(concept(X)).`, so würde gefragt werden, ob `t` wahr ist, obwohl `t,not(t)` natürlich nicht erfüllt werden kann. Das liegt daran, dass keines der beiden Teilziele `t` und `not(t)` für sich betrachtet immer fehlschlägt.

Eine weitere Möglichkeit die Anzahl der Fragen zu reduzieren ist herauszufinden, für welche Lösung am wenigsten Fragen gestellt werden müssen. Da die Anzahl der zu stellenden Fragen auch von den Antworten auf die Fragen selber abhängt, könnte man einen Mittelwert aus dem Minimum und dem Maximum der zu stellenden Fragen für die Lösung einer Anfrage bilden. Man würde dann versuchen die Anfrage, für die dieser Wert am niedrigsten ist, zuerst zu lösen. Ob das in Prolog mit vertretbarem Aufwand realisierbar ist, ist offen. Ein Nachteil dieses Ansatzes ist, dass man die Reihenfolge der Fragen nicht mehr durch die Reihenfolge der Klauseln in der Wissensbank „steuern“ kann, was in einigen Fällen wünschenswert sein könnte.

Ein in vielen Fällen sinnvolle Möglichkeit Wissen zu strukturieren, das Gliedern von Konzepten in Hierarchien, wird im nächsten Abschnitt beschrieben.

# 4. Implementierung eines Metainterpreters für Konzepthierarchien

Häufig werden in der Praxis Hierarchien (im Sinne von Klassifikationen) verwendet um einen einfacheren Überblick über komplexe Sachverhalte zu gewinnen. Beispiele dafür sind die Klassifizierung von Lebewesen, Kleidungsstücken und Fahrzeugen. Als einfaches, aber trotzdem realitätsnahes Beispiel wird in diesem Abschnitt eine kleine Wissensbasis zum Kopfschmerz verwendet, die in Quelltext B.2 (siehe Anhang) zu finden ist.

Da die bisher vorgestellten Metainterpreter universell verwendbar sind, kann man natürlich auch die Kopfschmerz-Wissensbank damit verarbeiten. Dabei gibt es jedoch einen negativen Nebeneffekt: Der Metainterpreter wird immer die allgemeinste Lösung schlussfolgern, d.h. im Fall der Kopfschmerz-Wissensbank wird die erste Lösung (falls es eine gibt) immer „`kopfschmerz`“ sein. Gewünscht wird in den meisten Fällen allerdings die speziellste Lösung, die möglich ist. Es wäre zum Beispiel viel zu ungenau ein Tier als „Säugetier“ zu klassifizieren.

## 4.1. Modifikation der Wissensbasis

Eine Hierarchie kann man als eine Baumstruktur sehen. Die Knoten wären dabei in unseren bisherigen Wissensbasen die Klauseln des Prädikats `concept`. Eine Möglichkeit sicherzustellen, dass nur gewünschte Lösungen erhalten werden, ist ein neues Prädikat einzuführen und die Wissensbasis damit zu erweitern. Die in B.3 definierte Wissensbasis ist auf diese Weise entstanden. Das neue Prädikat heist dort `diagnosis`. Im Körper der Klauseln dieses Prädikates sind, jeweils per Disjunktion verknüpft, das zugehörige Konzept und in negierter Form alle spezielleren Konzepte aufgelistet. Für das Konzept Kopfschmerz ergibt sich also folgende Klausel:

**Quelltext 4.1 (erweiterte Wissensbasis)**

```
diagnosis(P, kopfschmerz) :-
  concept(P, kopfschmerz),
  not(concept(P, spannungskopfschmerz)),
  not(concept(P, migraene)).
```

Der Hintergrund ist einfach, dass nur dann „`kopfschmerz`“ als Diagnose ausgegeben werden soll, wenn kein spezielleres Konzept erfüllt ist. Mit der Einführung eines weiteren Prädikates ist man sehr flexibel, d.h. man kann die Konzepte mehr oder weniger unabhängig von den zu stellenden Diagnosen definieren. Es ist laut Wissensbasis zum Beispiel nicht sinnvoll die Diagnose „`migraene`“ zu stellen, da man abhängig davon, ob das Teilziel `aura(P, migraene_mit_aura)` mit ja oder nein beantwortet wird immer die Diagnose „`migraene_mit_aura`“ bzw. „`migraene_ohne_aura`“ stellen kann. Man ist auch allgemein nicht gezwungen zu jedem Konzept auch eine Diagnose auszugeben. Diese Flexibilität hat ihren Preis darin, dass man eine größere Wissensbasis verwalten muss. Je nach Anwendung wäre es also wünschenswert, wenn ein

Metainterpreter Konzepthierarchien direkt unterstützen würde. Dass so etwas möglich ist, wird im nächsten Abschnitt gezeigt.

## 4.2. Direkte Unterstützung durch den Metainterpreter

Damit ein Metainterpreter für die Unterstützung von Konzepthierarchien effektiv implementiert werden kann werden einige Annahmen gemacht. Die Konzepte in der Wissensbasis sind im Prädikat `concept/2` gespeichert. Alle Anfragen haben die Form `agent(concept(X,Y))`, wobei `X` ein Bezeichner (z.B. der Name bzw. die Nummer des Patienten) und `Y` ein Merkmal (z.B. die Krankheit) ist. Bei der Konzepthierarchie, die als Baumstruktur aufgefasst werden kann, sollen nur die Blätter dieses Baumes mögliche Lösungen der Anfrage sein. Unter diesen Annahmen lässt sich ein geeigneter Metainterpreter konstruieren, der auf dem Metainterpreter aus Abschnitt 3.2 basiert. Die neu eingeführten bzw. geänderten Klauseln werden im Folgenden vorgestellt.

**Quelltext 4.2**
```
agent(Goal) :- retractall(answer(_)), presolve(Goal).
presolve(Goal) :- not(solve_sure_fail(Goal)), !, fail.
presolve(Goal) :- conceptsolve(Goal).
```

Im Prädikat `agent` ändert sich nur, dass statt `solve` die mit `conceptsolve` bezeichnete Konzept-Klauselkörperregel aufgerufen wird.

**Quelltext 4.3 (Konzept-Klauselkörperregel)**
```
conceptsolve(concept(X,Y)) :-
  clause(concept(X,Y), Body),
  not(rhs(concept(X,Y))),
  solve(Body).
conceptsolve(Goal) :-
  write('\nWarning: Henceforth the meta interpreter without \
         support for concept hierarchies is used!\n'),
  solve(Goal).
```

Interessant ist vor allem die erste Klausel der Konzept-Klauselkörperregel. Sie nimmt eine Anfrage der Form `concept(X,Y)` entgegen und sucht im Programm nach einer damit unifizierbaren Klausel. Danach wird über das Prädikat `rhs`, welches noch erklärt wird, sichergestellt, dass der gefundene Klauselkopf nirgendwo auf der rechten Seite einer beliebigen `concept`-Klausel im Programm vorkommt. Der Hintergrund ist, dass `concept`-Klauseln folgende Form haben, wenn damit eine Hierarchie aufgebaut werden soll:

**Quelltext 4.4**
```
concept(X, merkmal) :-
  concept(X, uebergeordnetes_merkmal1),
  concept(X, uebergeordnetes_merkmal2),
  [...]
```

Die Anzahl der übergeordneten Merkmale kann dabei auch 0 sein. Entscheident ist, dass nur die speziellsten Konzepte nicht auf der rechten Seite von `concept`-Klauseln vorkommen. Die Konzept-Klauselkörperregel ist also eine modifizierte Klauselkörperregel, die nur diejenigen `concept`-Klauseln als mögliche Lösungen betrachtet, deren Klauselköpfe nirgendwo auf der rechten Seite von im Programm definierten `concept`-Klauseln vorkommen. Sie dient praktisch gesehen als Filter vor der Anwendung von `solve`.

Falls die Anfrage nicht die richtige Form hat oder falls keine (weiteren) Konzepte mehr geschlussfolgert werden können, wird einfach `solve` aufgerufen, d.h. der Metainterpreter aus Abschnitt 3.2 verwendet. Wenn man vom Prolog-System also immer weitere Antworten verlangt (durch drücken von „`;`“), kann man vergleichen, welche Konzepte (und in welcher Reihenfolge) der Metainterpreter ohne die hier vorgestellten Modifikationen schlussfolgert.

**Quelltext 4.5**

```
rhs(concept(X,Y)) :-
  clause(concept(_,_), Body),
  contains(Body, concept(X,Y)).
```

Das in der Definition von `conceptsolve` verwendete `rhs` geht sequentiell durch alle `concept`-Klauseln im Programm (erste Zeile) und überprüft für jeden Klauselkörper, ob er das übergebene Argument `concept(X,Y)` enthält (zweite Zeile). `rhs` ist erfolgreich, falls `concept(X,Y)` in keinem Klauselkörper einer `concept`-Klausel vorhanden ist. Das Prädikat `contains` wird dabei folgendermaßen definiert:

**Quelltext 4.6**

```
contains((X1,X2), Y) :- contains(X1,Y); contains(X2,Y).
contains((X1;X2), Y) :- contains(X1,Y); contains(X2,Y).
contains(not(X), Y) :- X=Y.
contains(X, Y) :- X=Y.
```

`contains` nimmt eine syntaktische Zerlegung des ersten Arguments vor, wobei nur die vom Metainterpreter unterstützte Syntax beachtet wird. Falls `Y` im ersten Argument enthalten ist (geprüft wird über Unifikation mit Hilfe des in Prolog vordefinierten Prädikates `=`), ist die Anfrage erfolgreich, ansonsten nicht.

Der gesamte Quellcode des Metainterpreters mit Unterstützung für Konzepthierarchien lässt sich im Anhang unter Quelltext A.5 betrachten. Im Unterschied zu der in Abschnitt 4.1 vorgeschlagenen Modifikation der Wissensbasis geht bei diesem Ansatz Flexibilität verloren, dafür ist die Wissensbasis kleiner und einfacher zu verwalten. Zu beachten ist, dass wirklich nur die Blätter der durch die Konzepte gebildeten Baumstruktur als Lösung in Frage kommen. In der Wissensbasis B.2 für den Kopfschmerz wird also, falls das Konzept `kopfschmerz` erfüllt, aber die Konzepte `spannungskopfschmerz` und `migraene` nicht erfüllt sind, gar keine Lösung ausgegeben. Wenn man sich dieser Tatsache bewusst ist, stellt das jedoch keinen großen Nachteil dar, da man (falls das gewünscht ist) auf einfache Weise ein Blatt in der Baumstruktur erzeugen kann:

**Quelltext 4.7**

```
concept(P, einfacher_kopfschmerz) :-
  concept(P, kopfschmerz).
```

Insgesamt wurden im Beleg vier verschiedene Metainterpreter vorgestellt. Zuerst ein Standard-Metainterpreter als Basisgerüst (2), dann darauf aufbauend ein nachfragender Metainterpreter (3.1), der entscheidend bezüglich der Anzahl gestellter Fragen verbessert wurde (3.2). Als Abschluss wurde auf Konzepthierarchien eingegangen und neben der Möglichkeit der Modifikation der Wissensbasis (4.1) ein speziell darauf zugeschnitter Metainterpreter konstruiert (4.2).

# A. Komplette Quelltexte

**Quelltext A.1 (Metainterpreter aus Abschnitt 2)**

```
/**
 * Meta-Interpreter mit Konjunktion, Disjunktion und Negation.
 *
 * Aufruf: agent(concept(Bezeichner, Elefantenart))
 *
 * @author: Jens Lehmann
 */

% Metainterpreter wird als Agent betrachtet.
agent(Goal) :- solve(Goal).

% Negationsregel
solve(not(Goal)):-
  !,
  not(solve(Goal)).

% Konjunktionsregel
solve((Goal1,Goal2)):-
  !,
  solve(Goal1),
  solve(Goal2).

% Disjunktionsregel
solve((Goal1;_)) :- solve(Goal1), !.
solve((_;Goal2)) :- !, solve(Goal2).

% Boolesche Regel
solve(true) :- !.
solve(fail) :- !, fail.

% Klauselkörperregel
solve(Goal):-
  clause(Goal,Body),
  solve(Body).
solve(Goal):-
  clause(Goal, _), !, fail.

% Wissensbasis einbinden
:- include(wb_elefant).

% vorhandenes Wissen
ort(ele1, afrika).
ort(ele2, asien).
ort(ele3, afrika) :- fail.
```

```
ort(ele3, asien).
stosszahnrichtung_oben(ele1).
stosszahn_nicht_sichtbar(ele2).
stosszahn_nicht_sichtbar(ele3).
```

**Quelltext A.2 (Metainterpreter aus Abschnitt 2 mit Protokoll)**

```
/**
 * Meta-Interpreter mit Konjunktion, Disjunktion und Negation.
 * Inferenzschritte werden protokolliert.
 *
 * Aufruf: agent(concept(Bezeichner, Elefantenart))
 *
 * @author: Jens Lehmann
 */

% Metainterpreter wird als Agent betrachtet.
agent(Goal) :- solve(Goal).

% Negationsregel
solve(not(Goal)):-
  writef('Negationsregel    : not(%q) \n', [Goal]),
  !,
  not(solve(Goal)),
  writef('> Negationsregel erfolgreich auf not(%q) \
          angewendet \n', [Goal]).

% Konjunktionsregel
solve((Goal1,Goal2)):-
  writef('Konjunktionsregel  : (%q,%q) \n', [Goal1,Goal2]),
  !,
  solve(Goal1),
  solve(Goal2),
  writef('> Konjunktionsregel erfolgreich auf (%q,%q) \
          angewendet \n', [Goal1,Goal2]).

% Disjunktionsregel
solve((Goal1;Goal2)) :-
  writef('Disjunktionsregel  : (%q;%q) \n', [Goal1,Goal2]),
  solve(Goal1),
  writef('> Disjunktionsregel erfolgreich auf (%q;%q) \
          angewendet \n', [Goal1,Goal2]),
  !.
solve((Goal1;Goal2)) :-
  !, solve(Goal2),
  writef('> Disjunktionsregel erfolgreich auf (%q;%q) \
          angewendet \n', [Goal1,Goal2]).
```

```
% Boolesche Regel
solve(true) :-   writef('Boolesche Regel    : true \n'),!.
solve(fail) :-   writef('Boolesche Regel    : fail \n'),!, fail.

% Klauselkörperregel
solve(Goal):-
  clause(Goal,Body),
  writef('Klauselkörperregel : %q :- %q \n', [Goal,Body]),
  solve(Body),
  writef('> Klauselkörperregel erfolgreich auf %q :- %q \
          angewendet\n', [Goal,Body]).
solve(Goal):-
  clause(Goal, _), !, fail.

% Wissensbasis einbinden
:- include(wb_elefant).

% vorhandenes Wissen
ort(ele1, afrika).
ort(ele2, asien).
ort(ele3, afrika) :- fail.
ort(ele3, asien).
stosszahnrichtung_oben(ele1).
stosszahn_nicht_sichtbar(ele2).
stosszahn_nicht_sichtbar(ele3).
```

**Quelltext A.3 (Metainterpreter aus Abschnitt 3.1)**

```
/**
 * Meta-Interpreter mit Konjunktion, Disjunktion, Negation und
 * Fragen stellen bei nicht vorhandener Information.
 *
 * Aufruf: agent(concept(Bezeichner, Elefantenart))
 *
 * @author: Jens Lehmann
 */

% Metainterpreter wird als Agent betrachtet.
agent(Goal) :- retractall(answer(_)), solve(Goal).

% Negationsregel
solve(not(Goal)):-
  !,
  not(solve(Goal)).

% Konjunktionsregel
solve((Goal1,Goal2)):-
  !,
```

```
  solve(Goal1),
  solve(Goal2).

% Disjunktionsregel
solve((Goal1;_)) :- solve(Goal1), !.
solve((_;Goal2)) :- !, solve(Goal2).

% Boolesche Regel
solve(true) :- !.
solve(fail) :- !, fail.

% Klauselkörperregel
solve(Goal):-
  clause(Goal,Body),
  solve(Body).
solve(Goal):-
  clause(Goal, _), !, fail.

% Antwortregel
solve(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

% Frageregel
solve(Goal):-
  question(Goal,Answer),
  addanswer(Goal,Answer).

% eine Frage stellen (y und n als Antwort erlaubt)
question(Goal,Answer):-
  writef('Is %q true? (y/n) ', [Goal]),
  get_single_char(Answer),
  (Answer = 121; Answer = 110) ->
  (
    ((Answer = 121) -> print('yes'); print('no')),
    nl
  );
  write('Illegal answer, enter y for yes and n for no.'),
  nl, question(Goal, Answer).

% Eine Antwort auswerten. Die Antwort wird gespeichert.
% Fall 1: ja -> gibt true zurück
addanswer(Goal, 121) :- assert(answer(Goal)).
% Fall 2: nein -> schlägt fehl
addanswer(Goal, 110) :- assert(answer(Goal):-fail), fail.
```

```
% Wissensbasis einbinden
:- include(wb_elefant).
```

**Quelltext A.4 (Metainterpreter aus Abschnitt 3.2)**

```
/**
 * Meta-Interpreter mit Konjunktion, Disjunktion, Negation und
 * Fragen stellen bei nicht vorhandener Information.
 * Außerdem wurde die Anzahl zu beantwortender Fragen mit
 * Hilfe dreiwertiger Logik verringert.
 *
 * Aufruf: agent(concept(Bezeichner, Elefantenart))
 *
 * @author: Jens Lehmann
 */

% Metainterpreter wird als Agent betrachtet.
agent(Goal) :- retractall(answer(_)), presolve(Goal).
presolve(Goal) :- not(solve_sure_fail(Goal)), !, fail.
presolve(Goal) :- solve(Goal).

% Negationsregel
solve(not(Goal)):-
  !,
  not(solve(Goal)).

% erweiterte Konjunktionsregel
solve((Goal1,_)) :-
  not(solve_sure_fail(Goal1)),
  !, fail.
solve((_,Goal2)) :-
  not(solve_sure_fail(Goal2)),
  !, fail.
 solve((Goal1,Goal2)) :-
  solve_sure_success(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  !,
  solve(Goal1),
```

```
  solve(Goal2).

% erweiterte Disjunktionsregel
solve((Goal1;_)) :-
  solve_sure_success(Goal1),
  !.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
solve((Goal1;Goal2)) :-
  not(solve_sure_fail(Goal1)),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1;_)) :-
  solve(Goal1),
  !.
solve((_;Goal2)) :-
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
solve((_;Goal2)) :-
  !,
  solve(Goal2).

% Boolesche Regel
solve(true) :- !.
solve(fail) :- !, fail.

% Klauselkörperregel
solve(Goal):-
  clause(Goal,Body),
  solve(Body).
solve(Goal):-
  clause(Goal, _), !, fail.

% Antwortregel
solve(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

% Frageregel
solve(Goal):-
  question(Goal,Answer),
  addanswer(Goal,Answer).
```

```
% eine Frage stellen (y und n als Antwort erlaubt)
question(Goal,Answer):-
  writef('Is %q true? (y/n) ', [Goal]),
  get_single_char(Answer),
  (Answer = 121; Answer = 110) ->
  (
    ((Answer = 121) -> print('yes'); print('no')),
    nl
  );
  write('Illegal answer, enter y for yes and n for no.'),
  nl, question(Goal, Answer).

% Eine Antwort auswerten. Die Antwort wird gespeichert.
% Fall 1: ja -> gibt true zurück
addanswer(Goal, 121) :- assert(answer(Goal)).
% Fall 2: nein -> schlägt fehl
addanswer(Goal, 110) :- assert(answer(Goal):-fail), fail.

/**
 * solve_sure_fail/1
 * true: Anfrage wahr oder falsch
 * false: Anfrage falsch
 */

solve_sure_fail(not(Goal)):-
  !,
  not(solve_sure_success(Goal)).

solve_sure_fail((Goal1,Goal2)):-
  !,
  solve_sure_fail(Goal1),
  solve_sure_fail(Goal2).

solve_sure_fail((Goal1;_)) :- solve_sure_fail(Goal1), !.
solve_sure_fail((_;Goal2)) :- !, solve_sure_fail(Goal2).

solve_sure_fail(true) :- !.
solve_sure_fail(fail) :- !, fail.

solve_sure_fail(Goal):-
  clause(Goal,Body),
  solve_sure_fail(Body).
solve_sure_fail(Goal):-
  clause(Goal, _), !, fail.

solve_sure_fail(Goal):-
```

```
  clause(answer(Goal),_),
  !,
  answer(Goal).

solve_sure_fail(_).

/**
 * solve_sure_success/1
 * true: Anfrage wahr
 * false: Anfrage wahr oder falsch
 */

solve_sure_success(not(Goal)):-
  !,
  not(solve_sure_fail(Goal)).

solve_sure_success((Goal1,Goal2)):-
  !,
  solve_sure_success(Goal1),
  solve_sure_success(Goal2).

solve_sure_success((Goal1;_)) :- solve_sure_success(Goal1), !.
solve_sure_success((_;Goal2)) :- !, solve_sure_success(Goal2).

solve_sure_success(true) :- !.
solve_sure_success(fail) :- !, fail.

solve_sure_success(Goal):-
  clause(Goal,Body),
  solve_sure_success(Body).
solve_sure_success(Goal):-
  clause(Goal, _), !, fail.

solve_sure_success(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

% Wissensbasis einbinden
:- include(wb_elefant).
```

**Quelltext A.5 (Metainterpreter aus Abschnitt 4.2)**

```
/**
 * Meta-Interpreter mit Konjunktion, Disjunktion und Negation,
 * nachfragen von Informationen und Unterstützung von
 * Konzepthierarchien.
```

```
 *
 * Aufruf: agent(concept(Bezeichner, Krankheit))
 *
 * @author: Jens Lehmann
 */

% Metainterpreter wird als Agent betrachtet.
agent(Goal) :- retractall(answer(_)), presolve(Goal).
presolve(Goal) :- not(solve_sure_fail(Goal)), !, fail.
presolve(Goal) :- conceptsolve(Goal).

% Konzept-Klauselkörperregel
% lässt nur speziellste Konzepte zu
conceptsolve(concept(X,Y)) :-
  clause(concept(X,Y), Body),
  not(rhs(concept(X,Y))),
  solve(Body).
conceptsolve(Goal) :-
  write('\nWarning: Henceforth the meta interpreter without \
         support for concept hierarchies is used!\n'),
  solve(Goal).

% rhs(X) (right hand side)
% true: Konzept tritt irgendwo auf rechter Seite im Programm auf
% false: Konzept tritt niemals auf rechter Seite im Programm auf
rhs(concept(X,Y)) :-
  clause(concept(_,_), Body),
  contains(Body, concept(X,Y)).

% contains(X, Y)
% true: Y ist in X enthalten
% false: Y ist nicht in X enthalten
contains((X1,X2), Y) :- contains(X1,Y); contains(X2,Y).
contains((X1;X2), Y) :- contains(X1,Y); contains(X2,Y).
contains(not(X), Y) :- X=Y.
contains(X, Y) :- X=Y.

% Negationsregel
solve(not(Goal)):-
  !,
  not(solve(Goal)).

% erweiterte Konjunktionsregel
solve((Goal1,_)) :-
  not(solve_sure_fail(Goal1)),
  !, fail.
solve((_,Goal2)) :-
```

```
  not(solve_sure_fail(Goal2)),
  !, fail.
 solve((Goal1,Goal2)) :-
  solve_sure_success(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1,Goal2)) :-
  solve(Goal1),
  solve_sure_success(Goal2),
  !.
solve((Goal1,Goal2)) :-
  !,
  solve(Goal1),
  solve(Goal2).

% erweiterte Disjunktionsregel
solve((Goal1;_)) :-
  solve_sure_success(Goal1),
  !.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
solve((Goal1;Goal2)) :-
  not(solve_sure_fail(Goal1)),
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((Goal1;_)) :-
  solve(Goal1),
  !.
solve((_;Goal2)) :-
  not(solve_sure_fail(Goal2)),
  !, fail.
solve((_;Goal2)) :-
  solve_sure_success(Goal2),
  !.
solve((_;Goal2)) :-
  !,
  solve(Goal2).

% Boolesche Regel
solve(true) :- !.
solve(fail) :- !, fail.
```

```
% Klauselkörperregel
solve(Goal):-
  clause(Goal,Body),
  solve(Body).
solve(Goal):-
  clause(Goal, _), !, fail.

% Antwortregel
solve(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

% Frageregel
solve(Goal):-
  question(Goal,Answer),
  addanswer(Goal,Answer).

% eine Frage stellen (y und n als Antwort erlaubt)
question(Goal,Answer):-
  writef('Is %q true? (y/n) ', [Goal]),
  get_single_char(Answer),
  (Answer = 121; Answer = 110) ->
  (
    ((Answer = 121) -> print('yes'); print('no')),
    nl
  );
  write('Illegal answer, enter y for yes and n for no.'),
  nl, question(Goal, Answer).

% Eine Antwort auswerten. Die Antwort wird gespeichert.
% Fall 1: ja -> gibt true zurück
addanswer(Goal, 121) :- assert(answer(Goal)).
% Fall 2: nein -> schlägt fehl
addanswer(Goal, 110) :- assert(answer(Goal):-fail), fail.

/**
 * solve_sure_fail/1
 * true: Anfrage wahr oder falsch
 * false: Anfrage falsch
 */

solve_sure_fail(not(Goal)):-
  !,
  not(solve_sure_success(Goal)).

solve_sure_fail((Goal1,Goal2)):-
```

```
  !,
  solve_sure_fail(Goal1),
  solve_sure_fail(Goal2).

solve_sure_fail((Goal1;_)) :- solve_sure_fail(Goal1), !.
solve_sure_fail((_;Goal2)) :- !, solve_sure_fail(Goal2).

solve_sure_fail(true) :- !.
solve_sure_fail(fail) :- !, fail.

solve_sure_fail(Goal):-
  clause(Goal,Body),
  solve_sure_fail(Body).
solve_sure_fail(Goal):-
  clause(Goal, _), !, fail.

solve_sure_fail(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

solve_sure_fail(_).


/**
 * solve_sure_success/1
 * true: Anfrage wahr
 * false: Anfrage wahr oder falsch
 */

solve_sure_success(not(Goal)):-
  !,
  not(solve_sure_fail(Goal)).

solve_sure_success((Goal1,Goal2)):-
  !,
  solve_sure_success(Goal1),
  solve_sure_success(Goal2).

solve_sure_success((Goal1;_)) :- solve_sure_success(Goal1), !.
solve_sure_success((_;Goal2)) :- !, solve_sure_success(Goal2).

solve_sure_success(true) :- !.
solve_sure_success(fail) :- !, fail.

solve_sure_success(Goal):-
  clause(Goal,Body),
```

```
  solve_sure_success(Body).
solve_sure_success(Goal):-
  clause(Goal, _), !, fail.

solve_sure_success(Goal):-
  clause(answer(Goal),_),
  !,
  answer(Goal).

% Wissensbasis einbinden
:- include(wb_kopfschmerz).
```

# B. Wissensbasen

**Quelltext B.1 (Elefanten)**

```
% Wissensbasis für Elefantenarten
concept(T, afrikanischer_elefant) :-
  ort(T, afrika),
  stosszahnrichtung_oben(T).

concept(T, waldelefant) :-
  ort(T, afrika),
  not(stosszahnrichtung_oben(T)).

concept(T, asiatischer_elefant) :-
  ort(T, asien),
  (stosszahn_nicht_sichtbar(T);
  stosszahnrichtung_oben(T)).
```

**Quelltext B.2 (Kopfschmerzarten)**

```
% kleine Wissensbasis über Kopfschmerz von Dr. Petersohn
concept(P, kopfschmerz) :-
  lokalisation(P,schmerz, stirn);
  lokalisation(P,schmerz, auge);
  lokalisation(P,schmerz, schlaefe);
  lokalisation(P,schmerz, scheitel);
  lokalisation(P,schmerz, hinterkopf);
  lokalisation(P,schmerz, nacken);
  lokalisation(P,schmerz, untere_Halswirbelsäule).

concept(P, migraene) :-
  concept(P, kopfschmerz),
  (lokalisation(P, migraene, stirn);
   lokalisation(P, migraene, auge);
   lokalisation(P, migraene, schlaefe_beidseitig)),
  (schmerzqualitaet(P, migraene,pulsierend);
   schmerzqualitaet(P, migraene,drueckend);
   schmerzqualitaet(P, migraene,bohrend)),
  (schmerzintensitaet(P,migraene,mittel);
   schmerzintensitaet(P,migraene,stark)),
  (schmerzdauer(P,migraene,tage);
   schmerzdauer(P,migraene,stunden)),
  (schmerzhaeufigkeit(P,migraene,tage)),
  (zeitlicher_verlauf(P,migraene, attackenweise);
   zeitlicher_verlauf(P,migraene, zunahme_bei_phys_belastung)),
  (begleitsymptome(P,migraene,uebelkeit);
   begleitsymptome(P,migraene,erbrechen);
   begleitsymptome(P,migraene,lichtempfindlichkeit);
   begleitsymptome(P,migraene,laermempfindlichkeit);
   begleitsymptome(P,migraene,appetitlosigkeit);
```

```
   begleitsymptome(P,migraene,sehstoerung)).

concept(P, migraene_mit_aura) :-
  concept(P, migraene),
  aura(P, migraene_mit_aura).

concept(P, migraene_ohne_aura) :-
  concept(P, migraene),
  not(aura(P, migraene_mit_aura)).

concept(P, klassische_migraene) :-
  concept(P, migraene_mit_aura),
  aura_dauer(P, migraene_mit_aura,  minuten).

concept(P, spannungskopfschmerz) :-
  concept(P, kopfschmerz),
  (lokalisation(P, spannungskopfschmerz, stirn);
   lokalisation(P, spannungskopfschmerz, auge);
   lokalisation(P, spannungskopfschmerz, beidseitig)),
  (schmerzqualitaet(P, spannungskopfschmerz, dumpf);
   schmerzqualitaet(P, spannungskopfschmerz,drueckend);
   schmerzqualitaet(P, spannungskopfschmerz,klopfend);
   schmerzqualitaet(P, spannungskopfschmerz, haubenfoermig)),
  (schmerzintensitaet(P, spannungskopfschmerz, schwach);
   schmerzintensitaet(P, spannungskopfschmerz, mittel)),
  (trigger(P, spannungskopfschmerz, stress);
   trigger(P, spannungskopfschmerz, angst);
   trigger(P, spannungskopfschmerz, depression);
   trigger(P, spannungskopfschmerz, physische_Belastung)),
  (begleitsymptome(P,spannungskopfschmerz, uebelkeit);
   begleitsymptome(P,migraene,lichtempfindlichkeit)).
```

**Quelltext B.3 (Diagnosen zum Kopfschmerz (siehe Abschnitt 4.1))**

```
% kleine Wissensbasis über Kopfschmerz von Dr. Petersohn
concept(P, kopfschmerz) :-
  lokalisation(P,schmerz, stirn);
  lokalisation(P,schmerz, auge);
  lokalisation(P,schmerz, schlaefe);
  lokalisation(P,schmerz, scheitel);
  lokalisation(P,schmerz, hinterkopf);
  lokalisation(P,schmerz, nacken);
  lokalisation(P,schmerz, untere_Halswirbelsäule).

concept(P, migraene) :-
  concept(P, kopfschmerz),
  (lokalisation(P, migraene, stirn);
   lokalisation(P, migraene, auge);
```

```
   lokalisation(P, migraene, schlaefe_beidseitig)),
  (schmerzqualitaet(P, migraene,pulsierend);
   schmerzqualitaet(P, migraene,drueckend);
   schmerzqualitaet(P, migraene,bohrend)),
  (schmerzintensitaet(P,migraene,mittel);
   schmerzintensitaet(P,migraene,stark)),
  (schmerzdauer(P,migraene,tage);
   schmerzdauer(P,migraene,stunden)),
  (schmerzhaeufigkeit(P,migraene,tage)),
  (zeitlicher_verlauf(P,migraene, attackenweise);
   zeitlicher_verlauf(P,migraene, zunahme_bei_phys_belastung)),
  (begleitsymptome(P,migraene,uebelkeit);
   begleitsymptome(P,migraene,erbrechen);
   begleitsymptome(P,migraene,lichtempfindlichkeit);
   begleitsymptome(P,migraene,laermempfindlichkeit);
   begleitsymptome(P,migraene,appetitlosigkeit);
   begleitsymptome(P,migraene,sehstoerung)).

concept(P, migraene_mit_aura) :-
  concept(P, migraene),
  aura(P, migraene_mit_aura).

concept(P, migraene_ohne_aura) :-
  concept(P, migraene),
  not(aura(P, migraene_mit_aura)).

concept(P, klassische_migraene) :-
  concept(P, migraene_mit_aura),
  aura_dauer(P, migraene_mit_aura,  minuten).

concept(P, spannungskopfschmerz) :-
  concept(P, kopfschmerz),
  (lokalisation(P, spannungskopfschmerz, stirn);
   lokalisation(P, spannungskopfschmerz, auge);
   lokalisation(P, spannungskopfschmerz, beidseitig)),
  (schmerzqualitaet(P, spannungskopfschmerz, dumpf);
   schmerzqualitaet(P, spannungskopfschmerz,drueckend);
   schmerzqualitaet(P, spannungskopfschmerz,klopfend);
   schmerzqualitaet(P, spannungskopfschmerz, haubenfoermig)),
  (schmerzintensitaet(P, spannungskopfschmerz, schwach);
   schmerzintensitaet(P, spannungskopfschmerz, mittel)),
  (trigger(P, spannungskopfschmerz, stress);
   trigger(P, spannungskopfschmerz, angst);
   trigger(P, spannungskopfschmerz, depression);
   trigger(P, spannungskopfschmerz, physische_Belastung)),
  (begleitsymptome(P,spannungskopfschmerz, uebelkeit);
   begleitsymptome(P,migraene,lichtempfindlichkeit)).
```

```
% Diagnosen
diagnosis(P, kopfschmerz) :-
  concept(P, kopfschmerz),
  not(concept(P, spannungskopfschmerz)),
  not(concept(P, migraene)).
diagnosis(P, migraene_mit_aura) :-
  concept(P, migraene_mit_aura),
  not(concept(P, klassische_migraene)).
diagnosis(P, migraene_ohne_aura) :-
  concept(P, migraene_ohne_aura).
diagnosis(P, klassische_migraene) :-
  concept(P, klassische_migraene).
diagnosis(P, spannungskopfschmerz) :-
  concept(P, spannungskopfschmerz).
```

## Literatur

[1] Homepage SWI-Prolog http://www.swi-prolog.org/

Der Beleg wurde ohne Zuhilfenahme spezieller Literatur erstellt, sondern basiert auf dem vorhandenen Wissen zu Logik und Prolog. Die Wissensbasis zum Kopfschmerz wurde von Dr. Petersohn zur Verfügung gestellt.