Einführung in PROLOG II

PROLOG Syntax

Es gibt vier Arten von Termen:

- Atome
- Zahlen
- Variablen
- Komplexe Terme

Atome

1. uni, uni_hannover
2. ‘KI Kurs’, ‘§$”% 45’
3. @=, ====>

Variablen

1. X, Y
2. Variable, Output
3. _tag, _head
4. X_256

Matching

Wir wollen definieren, wann zwei Terme gleich sind (matchen). Warum ist dieses so wichtig?

Erstens ist es uns gestattet, komplexe Terme (also rekursiv strukturierte Terme) zu bauen, daher ist das Problem des matching sehr weitläufig, es muß z.B. die rekursive Struktur berücksichtigt werden.

Zweitens kann ein Term Variablen enthalten. Wenn wir z.B. *term1* und *term2* vergleichen, wobei *term1* eine Variable X enthält, ist es möglich durch geeignete Wahl des X die beiden Terme gleich zu machen. Uns interessiert am matchen also nicht nur, wann zwei Terme gleich sind, sondern auch, was wir mit den Variablen tun müssen, um sie gleich zu machen.

Das führt zu folgender Definition:

1. Seien *term1* und *term2* Konstanten. Dann **matchen** *term1* und *term2* wenn sie dasselbe Atom oder dieselbe Nummer sind.

2. Sei *term1* eine Variable und *term2* irgendein Term, dann **matchen** beide und *term1* wird zu *term2* instanziiert. Umgekehrt genauso.(Wenn Sie also beide Variablen sind, werden sie beide zueinander instaziiert „share values“)

3. Seien *term1* und *term2* komplex, dann **matchen** sie wenn
    a. Sie dasselbe Prädikatsymbol und dieselbe Anzahl von Argumenten haben **und**
    b. Alle ihre korrespondierenden Argumente matchen

4. Zwei Terme **matchen** wenn es aus den ersten drei Punkten folgt, dass sie **matchen**.

Die vierte Regel besagt, dass die ersten drei Punkte vollständig definieren, wenn zwei Terme matchen. Wenn zwei Terme aufgrund der ersten drei Regeln nicht matchen, denn matchen sie nie. Zum Beispiel matchen die Terme *batman* und *tochter(klein)* nicht. Warum? Der erste Term ist eine Konstante, der zweite Komplex, aber keine der ersten drei Regeln sagt uns, wie solche zwei Terme matchen.

Beispiele:

Wir benutzen das PROLOG-Prädikat **=/2** (/2 bedeutet: Das Prädikat braucht zwei Argumente)

Die Querie **: ?- =(mia,mia).** Liefert **yes**

Ebenso **?- mia = mia.**

Aber auch **?- 'mia' = mia.** Denn für PROLOG sind mia und 'mia' dasselbe Atom.

Beachte: **?- '2' = 2.** liefert **no**, denn2 ist eine Nummer, '2' ein Atom.

Was erwarten wir nach Regel 2 auf die Eingabe : **?- mia = X.** ?

Betrachten wir ein komplexeres Beispiel:

?- kill(shoot(gun),Y) = kill(X,stab(knife)).

Liefert:

X=shoot(gun)

Y=stab(knife)

Yes

Nach Regel 3 wird zuerst verglichen, ob die Terme dasselbe Prädikatsysmbol und dieselbe Anzahl von Argumenten haben. Da das gilt, wird mit Regel 2 versucht die Argumente zu matchen.

Frage: Was passiert Schritt für Schritt bei:

?- kill(shoot(gun), stab(knife)) = kill(X, stab(Y)). ?

Programmieren mit Matching

Wie schon gesagt, ist matching eine fundamentale Operation inPROLOG, es spielt eine wichtige Rolle im automatischen Beweisen, wie wir später sehen werden. Außerdem kann man einige nützliche Programme schreiben, indem man einfach durch komplexe Terme interessante Konzepte beschreibt. Mit Matching kann man dann die Information, die man haben will einfach erhalten.
Hier ist ein einfaches Beispiel nach Ivan Bratko. Die folgende zweizeilige Knowledge base definiert Grundsätzliches über Liniensegmente:

vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).

Auf den ersten Blick sieht dieses Beispiel sehr langweilig aus. Nur zwei Fakten, keine Regeln. Aber schauen wir genauer hin:
Wir haben ein **point/2** Prädikat, es repräsentiert die Cartesischen Koordinaten eines Punktes. **line/2** definiert aus den Punkten die Linie zwischen ihnen. Indem wir die Fähigkeit von PROLOG ausnutzen, komplexe Terme zu erstellen, können wir uns durch eine Hierarchie von Konzepten hocharbeiten.
vertical/2 beschreibt nun eine Linie, die durch zwei Punkte geht mit derselben X-Koordinate, **horizontal/2** eine Linie, die durch zwei Punkte geht, mit derselben Y-Koordinate.
Wie wenden wir dieses Wissen an?
?- vertical(line(point(1,1),point(1,3))).
yes
?- vertical(line(point(1,1),point(3,2))).
no
Das war zu erwarten. Aber wir können allgemeinere Fragen stellen:
?- horizontal(line(point(1,1),point(2,Y))).
Y=1
Unsere Querie war : Wenn wir eine horizontale Linie zwischen dem Punkt (1,1) und einem Punkt mit X-Koordinate 2 ziehen wollen, welche Y-Koordinate braucht dann der andere Punkt? PROLOG liefert uns die Antwort Y=1.
Betrachten wir nun:
?- horizontal(line(point(2,3),P)).
P = point(Z,3) ;
Die Querie war : Wenn wir eine horizontale Linie durch den Punkt (2,3) zeichnen wollen, welche anderen Punkte sind erlaubt? Die Antwort ist: Jeder Punkt, dessen Y-Koordinate 3 ist. Beachten Sie: Z ist eine Variable und daher Eclipse-PROLOGs Art uns mitzuteilen, dass die X-Koordinate beliebig sein kann.

Automatisches Beweisen (proof search)

Betrachten wir folgende Knowledge base:

f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X),g(X),h(X).

Was passiert, wenn wir folgende Querie starten:
?- k(X).

Die Antwort ist natürlich X=b, aber wie findet PROLOG dieses heraus?
Starten wir dazu den Tracer in Eclipse:
PROLOG liest die Knowledge Base und versucht zu matchen (*CALLl k(X)*). Das geht hier erst am Schluß: PROLOG hat nur eine Möglichkeit: Es muß k(X) mit dem Kopf der Regel **k(X) :- f(X),g(X),h(X).** matchen.
(Eigentlich ein Entscheidungspunkt: PROLOG hat sich entschieden die Querie mit dieser Regel zu matchen)
Dann ersetzt PROLOG die orginal Querie durch folgende Liste von goals:
f(X),g(X),h(X)
Um die erste Regel zu erfüllen (*CALL f(X))* kommt PROLOG an einen neuen Entscheidungspunkt, denn es erkennt, dass a und b die Regel erfüllen. PROLOG entscheidet sich für a (*EXIT f(a)*). Wenn PROLOG jetzt X durch a instanziiert, heißt die Liste der goals:
f(a),g(a),h(a)
PROLOG kann das zweite Subgoal erfüllen (Aufruf *CALL g(a)* und Ergebnis *EXIT g(a)*)
PROLOG findet dann aber, dass **h(a)** nicht erfüllbar ist (*FAIL h(...))*, und springt zum letzten Entscheidungspunkt zurück (REDO f(X)). Mit der neuen Instanziierung X durch b erhalten wir
f(b),g(b),h(b)
was erfüllbar ist, also ist X=b die Lösung.

Übungen:

1. Welche der folgenden Termpaare matchen? Wo möglich, geben Sie bitte die Variableninstanziierung an, die zu einem erfolgreichen matching führt.

    1. bread = bread
    2. 'Bread' = bread
    3. 'bread' = bread
    4. Bread = bread
    5. bread = sausage
    6. food(bread) = bread
    7. food(bread) = X
    8. food(X) = food(bread)
    9. food(bread,X) = food(Y,sausage)
    10. food(bread,X,beer) = food(Y,sausage,X)
    11. food(bread,X,beer) = food(Y,big_mac)
    12. food(X) = X
    13. meal(food(bread),drink(beer)) = meal(X,Y)
    14. meal(food(bread),X) = meal(X,drink(beer))

2. Wir haben folgende Knowledge Base

```
house_elf(dobby).
witch(hermione).
witch('McGonagall').
witch(rita_skeeter).
magic(X):-house_elf(X).
magic(X):-wizard(X).
magic(X):-witch(X).
```

Welche der folgenden Queries sind erfüllbar? Wo möglich, geben Sie bitte die Variableninstanziierung an, die zu einem Erfolg führt.

1. ?- magic(house_elf).
2. ?- wizard(harry).
3. ?- magic(wizard).
4. ?- magic('McGonagall').
5. ?- magic(Hermione).

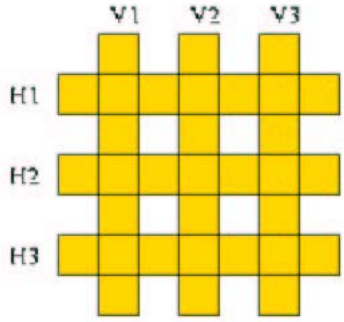
3. Wir haben hier ein (ausgesprochen kleines) Lexikon und eine Mingrammatik, die mit nur einer Regel definiert, dass ein Satz aus 5 Wörtern: Einem article, einem noun, einem verb, und wieder einem article und einem noun.

```
word(article,a).
word(article,every).
word(noun,wizard).
word(noun,quidditch_match).
word(verb,plays).
word(verb,likes).
```

```
sentence(Word1,Word2,Word3,Word4,Word5) :-
        word(article,Word1),
        word(noun,Word2),
        word(verb,Word3),
        word(article,Word4),
        word(noun,Word5).
```

Mit welcher Querie kann man herausfinden, welche Sätze diese Grammatik erzeugen? Listen Sie alle Sätze, die diese Grammatik erzeugen kann in der Reihenfolge auf, in der PROLOG sie generieren.

4. Wir haben hier 6 englische Wörter:
abalone, *abandon*, *anagram*, *connect*, *elegant*, *enhance*.
Sie sollen in diesem Kreuzworträtsel angeordnet werden.



Die folgende knowledge base repräsentiert ein Lexikon, das diese Wörter enthält.

```
word(abalone,a,b,a,l,o,n,e).
word(abandon,a,b,a,n,d,o,n).
word(enhance,e,n,h,a,n,c,e).
word(anagram,a,n,a,g,r,a,m).
word(connect,c,o,n,n,e,c,t).
word(elegant,e,l,e,g,a,n,t).
```

Schreiben Sie ein Prädikat crosswd/6 , das uns anzeigt, wie die Felder zu füllen sind. Die ersten drei Argumente könnten z.B. die vertikalen Wörter von links nach rechts sein, die weiteren drei die horizontalen von oben nach unten.