Einführung in PROLOG III

Rekursion

Betrachten wir die folgende Knowledge Base:

```
teurer(X,Y):-
        kostet_etwas_mehr(X,Y).
teurer(X,Y):-
        kostet_etwas_mehr(X,Z),
        teurer(Z,Y).

kostet_etwas_mehr(big_mac,pommes).
kostet_etwas_mehr(doener,big_mac).
```

Prädikate (hier *teurer*) können rekursiv auch definiert werden.
Betrachten wir die *deklarative* und die *prozedurale* Bedeutung dieser Regel.
Deklarativ (Was es aussagt):
Erste Regel: *Wenn* X etwas mehr als Y kostet, *dann* ist X teurer als Y
Zweite Regel: *Wenn* X etwas mehr als Z kostet *und* Z teurer als Y ist, *dann* ist X teurer als Y.
Prozedural (Was wirklich passiert):
Die erste Regel (Die Basisklausel) ist klar
Die zweite Regel gibt PROLOG eine alternative Strategie, um herauszufinden, ob X teurer ist als Y: Versuche ein Z zu finden, so dass X etwas mehr als Z kostet, und Z teurer als Y ist. Das heißt, Prolog zerlegt das Problem in zwei Teilprobleme.

Schauen wir uns an, was PROLOG mit folgender Query macht:

```
?- teurer(doener,pommes).
```

Zuerst versucht PROLOG die erste Regel anzuwenden. X wird mit doener und Y mit pommes gleichgesetzt, wodurch wir folgendes goal erhalten:

```
kostet_etwas_mehr(doener,pommes).
```

Aber die Knowledge Bas enthält diese Information nicht. Also versucht PROLOG die zweite Regel und erhält folgendes goal:

```
kostet_etwas_mehr(doener, Z),
teurer(Z, pommes).
```

Beim Blick in die Knowledge Base wird Z mit **big_mac** instanziiert, so dass der erste Teil erfüllt ist. Die erste Regel reduziert den zweiten Teil auf:

```
teurer(big_mac,pommes).
```

Und das ist ein Fakt in der Knowledge Base.

Es ist einleuchtend, dass man beim Schreiben eines rekursiven Prädikates immer wenigstens zwei Klauseln benötigt: Eine Basisklausel, die die Rekursion beenden kann, und eine, die die Rekursion enthält.

Beispiel Nachkomme

Da wir jetzt wissen, was Rekursion für PROLOG bedeutet, betrachten wir jetzt, warum Rekursion so wichtig ist. Betrachten wir folgende Knowledge Base:

```
kind(jan,alfred).
kind(jonathan,jan).
```

Wenn wir jetzt die Beziehung *Nachkomme* definieren wollen, können wir das nicht-rekursiv tun:

```
nachkomme(X,Y):-
        kind(X,Y).
nachkomme(X,Y):-
        kind(X,Z),
        kind(Z,Y).
```

Wenn wir aber mehr als zwei Generationen betrachten wollten, müssten wir immer weitere Regeln anhängen, daher bietet sich die rekursive Version an :

```
nachkomme(X,Y):-
        kind(X,Y).
nachkomme(X,Y):-
        kind(X,Z),
        nachkomme(Z,Y).
```

Beispiel Sukzessor

Es gibt verschieden Arten Zahlen zu schreiben. Eine begegnet uns in der Logik:

1. 0 ist eine Zahl
2. Wenn *X* eine Zahl ist, dann auch *succ(X)*.

Succ steht für Sukzessor, also ist succ(X) die Zahl, die man erhält, wenn man zu X Eins addiert. Diese Definition wird zu einem PROLOG-Programm mit folgender Knowledge Base:

```
zahl(0).
zahl(succ(X)):-zahl(X).
```

Natürlich ergibt die Query

```
?- zahl (succ(succ(succ(0)))).
```

Ein yes.
Aber interessanter ist:

```
?- zahl(X).
```

Also: „Zeig mir einige Zahlen!“ ergibt:

X = 0 ;
X = succ(0) ;
X = succ(succ(0)) ;
X = succ(succ(succ(0))) ;
X = succ(succ(succ(succ(0)))) ;
X = succ(succ(succ(succ(succ(0))))) ;
X = succ(succ(succ(succ(succ(succ(0)))))) ;
X = succ(succ(succ(succ(succ(succ(succ(0)))))))

PROLOG zählt auf eine besondere Art und Weise, indem es Zahlen mithilfe von matching konstruiert.

Vorsicht!
Es ist wichtig, zu verstehen, wie PROLOG vorgeht, indem es versucht Queries zu beantworten. Was passiert z.B., wenn wir obige Knowledge Base in folgener Form schreiben:

```
zahl(succ(X)):-zahl(X).
zahl(0).
```

Und dann die Querie

```
?- zahl(X).
```

starten?

Übungen:

1. Betrachten wir folgende Knowledge Base:

```
bigger(cat,mouse).
bigger(dog,cat).
bigger(sheep,dog).
bigger(horse,sheep).
bigger(elephant,horse).
```

Schreiben Sie ein rekursives Prädikat biggerThan/2 welches alle "Größer-Als"-Beziehungen zwischen den Tieren herausbekommt, dass also ein Elefant größer ist als eine Maus, usw.

2. Betrachten wir folgende Knowledge Base:

```
directTrain(lueneburg,hamburg).
directTrain(bremen,lueneburg).
directTrain(braunschweig,hannover).
directTrain(hannover,lueneburg).
directTrain(hamburg,kiel).
directTrain(magdeburg,braunschweig).
directTrain(berlin,magdeburg).
```

Sie enthält Informationen über direkte Zugverbindungen. Natürlich kann man auch reisen, indem man mehrere Züge hintereinander nimmt. Schreiben Sie ein rekursives Prädikat travelBetween/2 welches uns mitteilt, ob man zwischen zwei Orten fahren kann. Zum Beispiel sollte die Querie:

```
travelBetween(berlin,hamburg).
```

`yes' ergeben.

Gehen wir jetzt davon aus, dass eine Verbindung von A nach B auch eine Verbindung von B nach A bedeutet. Ergänzen Sie eine Rule, die dieses ermöglicht, und testen Sie sie mit der Querie:

```
travelBetween(hamburg,berlin).
```

3. Betrachten wir folgende Knowledge Base:

```
byCar(auckland,hamilton).
byCar(hamilton,raglan).
byCar(valmont,saarbruecken).
byCar(valmont,metz).

byTrain(metz,frankfurt).
byTrain(saarbruecken,frankfurt).
```

```
byTrain(metz,paris).
byTrain(saarbruecken,paris).

byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(losAngeles,auckland).
```

Schreibe Sie ein Prädikat travel/2, welches herausfindet, ob es möglich ist von einem Ort zum anderen zu reisen, indem man Auto, Zug oder Flugzeug nimmt. Zum Beispiel sollte `yes' auf die Querie travel(valmont,raglan) geliefert werden.

- travel/2 sagt uns also, dass es möglich ist, von Vamont nach Raglan zu reisen. Schreiben Sie ein Prädikat travel/3, welches die exakte Route angibt. Auf die Querie travel(valmont,paris,go(valmont,metz,go(metz,paris))) sollte also ‘yes’ und X = go(valmont,metz,go(metz,paris,go(paris,losAngeles))) auf die Querie travel(valmont,losAngeles,X). folgen

- Erweitern Sie travel/3 so, dass sie nicht nur die Route liefert, sondern auch wie von einem Ort zum anderen gereist wird, also car, train, plane.