Datalog/Prolog

Deduktion mit Datalog/Logikprogrammierung

Tobias Buhr

tobiasbuhr@gmx.de

01. März 2005

Inhalt

Einführung
Klassifikation von Programmiersprachen

Imperative Sprachen
- Folge von nacheinander ausgeführten Anweisungen

Prozedurale Sprachen
- Variablen, Zuweisungen, Kontrollstrukturen

Objektorientierte Sprachen
- Objekte und Klassen
- ADT und Vererbung

Deklarative Sprachen
- Spezifikation dessen, was berechnet werden soll

Funktionale Sprachen
- keine Seiteneffekte
- Rekursion

Logische Sprachen
- Regeln zur Definition von Relationen

Inhalt

Deduktion mit Datalog

- Datalog ermöglicht die deduktive Erweiterung einer relationalen DB
- auf Basis eines eingeschränkten Prädikatenkalküls
- Faktenbasis
- Regeln
- gefolgertes Wissen

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Anfrage:

```
?-stadt(X).
```

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Anfrage:

```
?-stadt(X).
```

Ergebnis:

```
X=aachen
```

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Anfrage:

```
?-stadt(X).
```

Ergebnis:

```
X=aachen ;
X=bonn
```

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Anfrage:

```
?-stadt(X).
```

Ergebnis:

```
X=aachen ;
X=bonn ;
no
```

Datalog - kleines Beispiel(1)

Fakten (Wissen):

```
stadt(aachen).
stadt(bonn).
```

Anfrage:

```
?-stadt(X).
```

Ergebnis:

```
X=aachen ;
X=bonn ;
no
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Anfrage:

```
?-stadtInDeutschland(S).
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Anfrage:

```
?-stadtInDeutschland(S).
```

Ergebnis:

```
S=aachen
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Anfrage:

```
?-stadtInDeutschland(S).
```

Ergebnis:

```
S=aachen ;
S=bonn
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Anfrage:

```
?-stadtInDeutschland(S).
```

Ergebnis:

```
S=aachen ;
S=bonn ;
no
```

Datalog - kleines Beispiel(2)

Fakten (Wissen):

```
stadtInNrw(aachen).
stadtInNrw(bonn).
```

Regel:

```
stadtInDeutschland(S):-stadtInNrw(S).
```

Anfrage:

```
?-stadtInDeutschland(S).
```

Ergebnis:

```
X=aachen ;
X=bonn ;
no
```

Fakten

- (Datalog-)Fakten
- entsprechen Tupeln von Relationen
- Beispiele
  - $stadt(aachen)$
  - $hauptstadt(bonn, deutschland)$

(Datalog-)Regeln

- (Datalog-)Regeln
- basieren auf Prädikatenkalkül
- Schreibweise in Prolog:
  $p(...) :- p_1(...), ..., p_k(...).$
  $father(X) :- parent(X), male(X).$
- Regeln sind Implikationen

Anfragen

- Anfrage (Ziel, goal)
- Beispiel
  - Schreibweise in Prolog: $? - father(otto, X)$

Beispiel Mortal

Mortal.pl:

$$mortal(X) :- person(X).$$
$$person(socrates).$$

Beispiel Mortal

Mortal.pl:

$$mortal(X) :- person(X).$$
$$person(socrates).$$

Anfrage:

$$?- mortal(socrates).$$

Beispiel Mortal

Mortal.pl:

$$mortal(X) :- person(X).$$
$$person(socrates).$$

Anfrage:

$$?- mortal(socrates).$$

Antwort:

$$yes$$

Beantworten von Anfragen

Resolution, Unifikation

- Herleiten der Antworten mit einem logischen Beweisverfahren
- Beweisidee: Behaupte Unerfüllbarkeit der Anfrage und finde Gegenbeispiele
- Resolution (Robinson 1965)
  - Verbinde zwei Klauseln, die dasselbe Literal, aber mit verschiedenen Vorzeichen enthalten
  - vollständiges und korrektes Verfahren zum Ableiten von Widersprüchen in Klauselmengen
- Unifikation: Zwei atomare Formeln, bzw. Terme unifizieren, falls:
  - sie syntaktisch gleich sind, oder
  - falls ihre Variablen so belegt werden können, dass eine Substitution existiert, so dass beide Terme syntaktisch gleich werden.

Anfrage an Mortal.pl

- Ist Socrates sterblich?

$$\frac{?- \mathit{mortal}(\mathit{socrates}) \qquad \text{Anfrage}}{\mathit{mortal}(X) :- \mathit{person}(X) \qquad \text{Regel}} \qquad \text{Resolvente}$$
$$\mathit{person}(\mathit{socrates})$$

Klauseln müssen durch geeignete Substitution unifiziert werden

- Dahinter steckt die allgemeine Schlussregel:
$((A \rightarrow B) \wedge (B \rightarrow C)) \rightarrow (A \rightarrow C)$

Backtracking
Ancestor-Beispiel (1)

ancestor.pl:

```
anc(X,Y):-par(X,Y).          par(otto,anna).
anc(adam,X):-person(X).      par(berta,otto).
                             par(berta,email).

person(X):-lives(X,Y).       lives(otto,hamburg).
                             lives(anna,kiel).
```

Backtracking
Ancestor-Beispiel (1)

ancestor.pl:

```
anc(X,Y):-par(X,Y).          par(otto,anna).
anc(adam,X):-person(X).      par(berta,otto).
                             par(berta,email).

person(X):-lives(X,Y).       lives(otto,hamburg).
                             lives(anna,kiel).
```

- Anfrage: $? - anc(X, anna).$

Backtracking
Ancestor-Beispiel (1)

ancestor.pl:

```
anc(X,Y):-par(X,Y).          par(otto,anna).
anc(adam,X):-person(X).      par(berta,otto).
                             par(berta,email).

person(X):-lives(X,Y).       lives(otto,hamburg).
                             lives(anna,kiel).
```

- Anfrage: $? - anc(X, anna).$
- $X = otto$

Backtracking
Ancestor-Beispiel (1)

ancestor.pl:

```
anc(X,Y):-par(X,Y).          par(otto,anna).
anc(adam,X):-person(X).      par(berta,otto).
                             par(berta,email).

person(X):-lives(X,Y).       lives(otto,hamburg).
                             lives(anna,kiel).
```

- Anfrage: $? - anc(X, anna).$
- $X = otto$
- Backtracking ...!

Backtracking
Ancestor-Beispiel (1)

ancestor.pl:

```
anc(X,Y):-par(X,Y).          par(otto,anna).
anc(adam,X):-person(X).      par(berta,otto).
                             par(berta,email).

person(X):-lives(X,Y).       lives(otto,hamburg).
                             lives(anna,kiel).
```

- Anfrage: $? - anc(X, anna).$
- $X = otto$
- Backtracking ...!
- $X = adam$

Rekursion
Ancestor-Beispiel (2)

ancestor.pl:

```
anc(X,Y):-par(X,Y).                 par(otto,anna).
anc(adam,X):-person(X).             par(berta,otto).
anc(X,Y):-par(Z,Y),anc(X,Z).        par(berta,email).

person(X):-lives(X,Y).              lives(otto,hamburg).
                                    lives(anna,kiel).
```

Rekursion
Ancestor-Beispiel (2)

ancestor.pl:

```
anc(X,Y):-par(X,Y).                 par(otto,anna).
anc(adam,X):-person(X).             par(berta,otto).
anc(X,Y):-par(Z,Y),anc(X,Z).        par(berta,email).

person(X):-lives(X,Y).              lives(otto,hamburg).
                                    lives(anna,kiel).
```

- Rekursion: $anc(X, Y) : -anc(X, Z), par(Z, Y).$

Rekursion
Ancestor-Beispiel (2)

ancestor.pl:

```
anc(X,Y):-par(X,Y).                 par(otto,anna).
anc(adam,X):-person(X).             par(berta,otto).
anc(X,Y):-par(Z,Y),anc(X,Z).        par(berta,email).

person(X):-lives(X,Y).              lives(otto,hamburg).
                                    lives(anna,kiel).
```

- Rekursion: $anc(X, Y) :- anc(X, Z), par(Z, Y).$
- Anfrage: $? - anc(X, anna).$

Rekursion
Ancestor-Beispiel (2)

ancestor.pl:

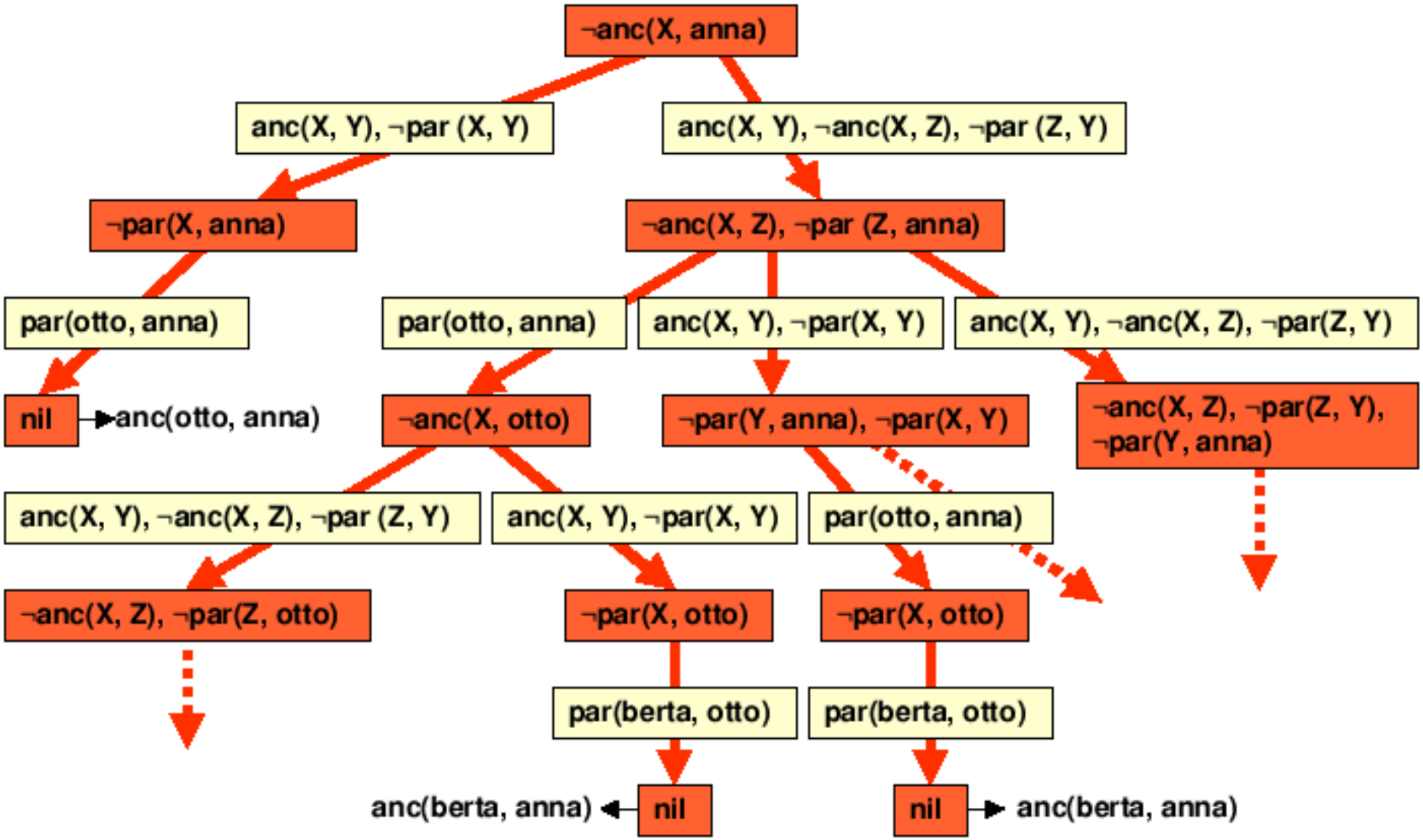
```
anc(X,Y):-par(X,Y).                 par(otto,anna).
anc(adam,X):-person(X).             par(berta,otto).
anc(X,Y):-par(Z,Y),anc(X,Z).        par(berta,email).

person(X):-lives(X,Y).              lives(otto,hamburg).
                                    lives(anna,kiel).
```

- Rekursion: $anc(X, Y) :- anc(X, Z), par(Z, Y).$
- Anfrage: $? - anc(X, anna).$
- Backtracking...!

Rekursion, Backtracking

Ancestor-Beispiel (3)



Inhalt

Zusammenfassung

- Fakten $stadt(koeln)$.
- Datalog-Regeln $stadtInDL(X) :- stadtInNrw(X)$.
- Anfragen $? - mortal(X)$.
- Widerspruchsbeweis, Resolution, Unifikation
- Backtracking
- Rekursion

Zusammenfassung
Notation

Variable:	Zeichenkette, die mit Großbuchstaben beginnt $X, X1, Haus$
Konstante:	Zahl/Zeichenkette, die mit Kleinbuchstaben beginnt $eins, 1, bauhaus$
Prädikat:	Zeichenkette, die mit Kleinbuchstaben beginnt p, add
Term:	Konstante oder Variable $bauhaus, X$
Atom:	n-stelliges Prädikat angewandt auf n Terme $p/2 : p(eins, zwei)$
Klausel:	Liste von Literalen $\{p(X), \neg q(X)\}$

Datalog/Prolog
Allgemeines

- Datalog $\subset$ Prolog
- Datalog ermöglicht die deduktive Erweiterung einer Relationalen DB
- Prolog: Interpreter für Logikprogramme
- SWI-Prolog (frei)
-  *http://www.swi-prolog.org*
  - Referenz
  - links zu Tutorials und Beispiel-Programmen
- SICStus-Prolog (kommerziell)

 *http://www.sics.se/sicstus*

Inhalt

Ende (Ihr seid dran!)

- Aufgabe 1
  - Fakten,Regeln
  - Anfragen
  - Datalog-Programm

Aufgabe 1 (1)

vatermutter.pl

```
weiblich(ada)                    maennlich(fritz).
weiblich(eva)                    maennlich(hans).
weiblich(maria)                  maennlich(heinz).
weiblich(ana)                    maennlich(otto).
weiblich(petra)                  maennlich(paul).
                                 maennlich(peter).

istMutterVon(ada, ana).          verheiratet(ada, hans).
istMutterVon(eva, paul).         verheiratet(eva, heinz).
istMutterVon(Maria,petra).       verheiratet(maria, fritz).
istMutterVon(Maria, peter).
istMutterVon(ana, otto).

istVaterVon(V,K):- verheiratet(M,V), istMutterVon(K,M).
istOnkelVon(O,K):- maennlich(O), istMutterVon(G,O),
                                 istMutterVon(G, ), istVaterVon( , ).
```

Aufgabe 1 (2)
Fakten, Regeln, Anfragen

1. Identifiziere Fakten und Regeln in *vatermutter.pl*.
2. Finde und korrigiere die Fehler in *vatermutter.pl*.
3. Erweitere *vatermutter.pl* um
   - 3.1 ada ist Mutter von Fritz.
   - 3.2 ana ist verheiratet mit paul.
   - 3.3 die fehlenden Variablen in $istOnkelVon(...) :- ....$
   - 3.4 jemand ist eine Tante, wenn ... .
4. Stelle 2 beliebige Anfragen.

Aufgabe 1 (1)

vatermutter.pl

```
weiblich(ada).                  maennlich(fritz).
weiblich(eva).                  maennlich(hans).
weiblich(maria).                maennlich(heinz).
weiblich(ana).                  maennlich(otto).
weiblich(petra).                maennlich(paul).
                                maennlich(peter).
% arg1 istMutterVon arg2       % arg1 ist verheiratet mit arg2
istMutterVon(ada, ana).         verheiratet(ada, hans).
istMutterVon(eva, paul).        verheiratet(eva, heinz).
istMutterVon(maria,petra).      verheiratet(maria, fritz).
istMutterVon(maria, peter).     verheiratet(ana, paul).
istMutterVon(ana, otto).
istMutterVon(ada, fritz).
istVaterVon(V,K):- verheiratet(M,V), istMutterVon(M,K).
istOnkelVon(O,K):- maennlich(O), istMutterVon(G,O),
                                 istMutterVon(G,B), istVaterVon(B,K).
```

Inhalt

Einführung

- "echte" /"volle" Logikprogrammierung
- Spezialprädikate (Listen-Konstruktion)
- Terme als Datenstrukturen

Inhalt

Terme als Datenstrukturen

- Konstruktorsymbol
- Bsp.: nat. Zahlen: $0, suc(0), suc(suc(0)), ...$
- Addition:

Rekursive Definition (math.): $x + 0 = 0$

$(x + y') = (x + y)'$

Prolog:

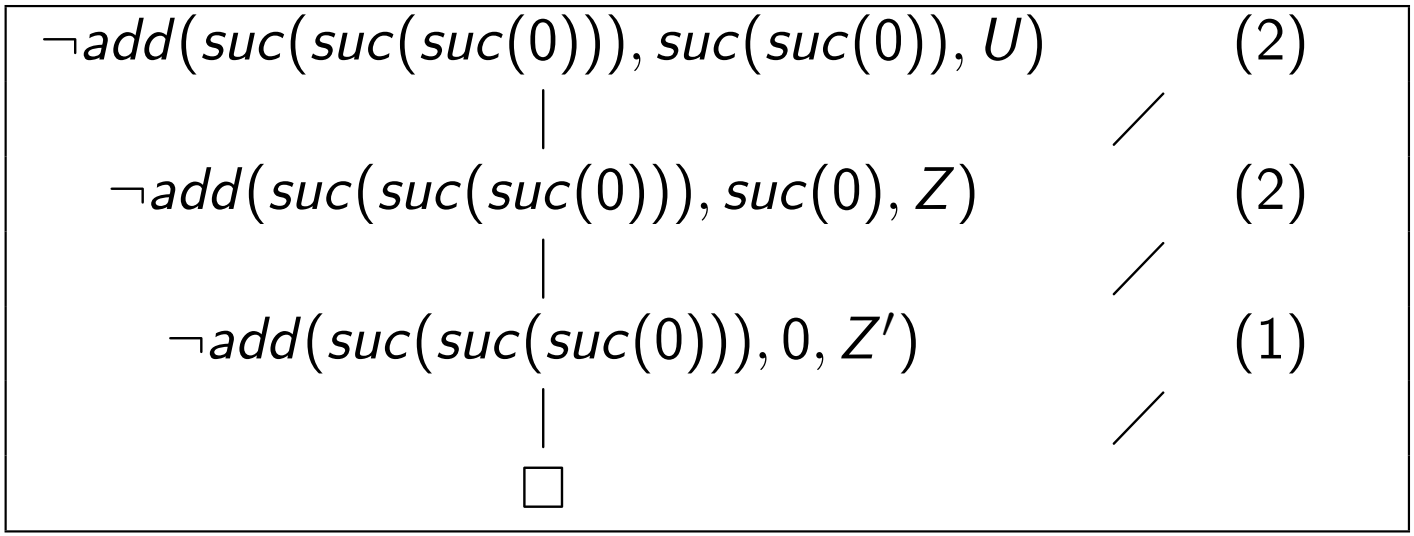
$add(X, 0, X).$

$add(X, suc(Y), suc(Z)) :- add(X, Y, Z).$

Addition

Berechne $3 + 2$

- Anfrage: $? - add(suc(suc(suc(0))), suc(suc(0)), U)$
- Programmklauseln
  $(1)\{add(X, 0, X)\}$
  $(2)\{add(X, suc(Y), suc(Z)), \neg add(X, Y, Z)\}$
- Resolution mit Anfrage:



- Substitutionen: $[U/suc(Z)], [Z/suc(Z')], [Z'/suc(suc(suc(0)))]$
- $U = suc(suc(suc(suc(suc(0)))))$

Inhalt

Listen

- Liste: verschachtelte Datenstruktur
- Konstruktor: $cons(x, y)$
- Prolog bietet abkürzende Schreibweise
  - $[a_1, a_2, ..., a_k]$ steht für $cons(a_1, cons(a_2, ...cons(a_k, nil)))$
  - $[x|y]$ steht für $cons(x, y)$
  - [] steht für die leere Liste *nil*
- Beispiel: $[a, b, c]$ steht für $cons(a, cons(b, cons(c, nil)))$
- verschachtelte Listen

Listen - Hilfsprädikate

- Einfügen: *insert*

$$insert(X, Xs, [X|Xs]).$$

- Listen-Konkatenation: *append*

$$append([], Xs, Xs).$$
$$append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).$$

- enthalten-sein: *member*

$$member(X, [X|\_]).$$
$$member(X, [\_|Ys]) :- member(X, Ys).$$

Listen - Permutation

- Permutieren:

$$permut([X],[X]).$$
$$permut([X|Xs],Zs) :- permut(Xs,Ts), nd\_ins(X,Ts,Zs).$$

- dafür benötigt: nicht-deterministisches Einfügen

$$nd\_ins(X,Xs,[X|Xs]).$$
$$nd\_ins(X,[Y|Ys],[Y|Zs]) :- nd\_ins(Xs,Ys,Zs).$$

Inhalt

Zusammenfassung

- Terme als Datenstrukturen $suc(suc(...suc(0)...))$
- Listen $[a, b, c]$
- Reihenfolge der Anweisungen spielt u.U. eine Rolle

Inhalt

Ende - Ihr seid dran!

- Aufgabe 1
  - Listen
- Aufgabe 2
  - Strings

Aufgabe 1

1. Listen
   - 1.1 Forme in Prolog-Listendarstellung um: $cons(a, cons(b, cons(c, nil)))$.
   - 1.2 Wofür steht folgende abkürzende Bezeichnung: $[[a, [b, c]]|[d, e]]$?
   - 1.3 Erstelle auf der Basis von $permut(...)$ einen Sortieralgorithmus für Listen.

Aufgabe 2

1. Was könnte die Anfrage $? - concat('hallo','Welt',X).$ bewirken?
2. Benutze *concat* um das Präfix von *Test* in *getTest* zu bestimmen.
3. Benutze *concat* um alle Suffixe von *get* in der Liste ['*getInstance*','*getDefault*','*getChildren*'] zu bestimmen.