

Prolog

11. und 12. Kapitel: fortgeschrittene Prologprädikate

Dozentin: Wiebke Petersen

Kursgrundlage: Learn Prolog Now (Blackburn, Bos, Striegnitz)

Manipulation der Wissensbasis

Prolog stellt Prädikate zur Verfügung, die eine Manipulation der Wissensbasis während der Laufzeit eines Programms ermöglichen: `assert/1`, (`asserta/1`, `assertz/1`), `retract/1`, `retractall/1`

`assert/1`: `assert(Clause)` fügt die Klausel `Clause` zur Wissensbasis dynamisch hinzu.

`asserta(Clause)` fügt die Klausel als neue erste Klausel hinzu.

`assertz(Clause)` fügt die Klausel als neue letzte Klausel hinzu.

`retract/1`!: `retract(Clause)` löscht eine Klausel, die mit `Clause` matcht, aus der Wissensbasis.

`retractall/1`!: `retractall(Clause)` löscht alle Klauseln, die mit `Clause` matchen.

Zusammenfassung Kapitel 10

- Wir haben das Prädikat `fail/0` kennengelernt, das immer scheitert.
- Wir haben den Cut kennengelernt und gesehen, wie man Negation in Prolog als „negation as failure“ definieren kann.
- Wir haben gelernt zwischen roten und grünen Cuts zu unterscheiden.
- **Wichtig:** Cuts zerstören die Deklarativität von Prologprogrammen und sollten daher mit Bedacht eingesetzt werden.
- **Keywords:** „negation as failure“, roter und grüner Cut, Cut-Fail-Kombination
- **Ausblick Kapitel 11 und 12:** Manipulation der Wissensbasis, Sammlung aller Lösungen einer Anfrage, Dateien lesen und schreiben.

Manipulation der Wissensbasis: Beispiel (1)

Hinzufügen von Klauseln

```
?- assert(mensch(anna)).
true.
?- asserta(mensch(tom)).
true.
?- assert(mensch(mia)).
true.
?- assertz(mensch(otto)).
true.
?- listing(mensch/1).
:- dynamic mensch/1.
mensch(tom).
mensch(anna).
mensch(mia).
mensch(otto).
true.
```

Löschen von Klauseln

```
?- retract(mensch(mia)).
true.
?- retract(mensch(X)).
X = tom.
true.
?- listing(mensch/1).
:- dynamic mensch/1.
mensch(anna).
mensch(otto).
true.

?- retractall(mensch(X)).
true.
?- listing(mensch/1).
:- dynamic mensch/1.
true.
```

Manipulation der Wissensbasis: Beispiel (2)

Im Prinzip ist es mit den Manipulationsprädikaten möglich, direkt von der Konsole zu programmieren:

```
?- assert(sterblich(X):- mensch(X)).
true.
?- assert(mensch(mia)).
true.
?- assert(sterblich(garfield)).
true.

?- sterblich(X).
X = mia;
X = garfield.
```

Dies ist jedoch nur selten eine gute Idee, da sie genau aufpassen müssen, in welcher Reihenfolge sie die Klauseln einfügen.

copy_term/2: Anwendung von assert/1 und retract/

`copy_term(Term1,Term2)` erstellt in `Term2` eine Kopie von `Term1` mit neuen Variablen.

```
% Praedikate, die waehrend der Laufzeit veraendert werden sollen,
% muessen als dynamisch deklariert werden.
:- dynamic temporaer/1.
copy_term(T1,T2):-
    assert( temporaer(T1)),
    retract(temporaer(T2)).
```

```
?- copy_term(vorfahr(A,B),Z).
Z = vorfahr(_G1, _G2).
```

```
?- copy_term(vorfahr(A,B),Z1),copy_term(vorfahr(A,B),Z2).
Z1 = vorfahr(_G1, _G2),
Z2 = vorfahr(_G3, _G4).
```

Programmierung durch Memoisierung

Die Programmierertechnik Ergebnisse in der Wissensbasis zu speichern, um bei späteren Berechnungen darauf zurück greifen zu können, heißt **Memoisierung**.

Beispiel: Fibonacci-Zahlen ohne Memoisierung

$fib(0) = 0$, $fib(1) = 1$, $fib(n) = fib(n-2) + fib(n-1)$

```
fib(0,0). % fib(0)=0
fib(1,1). % fib(1)=1

% fib(n)=fib(n-1)+fib(n-2)
fib(N,Res) :-
    N>1,
    N1 is N-1,
    N2 is N-2,
    fib(N1,R1),
    fib(N2,R2),
    Res is R1+R2.
```

Testen sie die folgenden Anfragen:

```
?- fib(4,R).
?- fib(25,R).
?- fib(40,R).
```

Warum ist das Programm so langsam?

Programmierung durch Memoisierung: Fibonacci

Fibonacci-Zahlen mit Memoisierung

$fib(0) = 0$, $fib(1) = 1$, $fib(n) = fib(n-2) + fib(n-1)$

```
% fib/2 muss als dynamisches
% Praedikat deklariert werden:
:- dynamic(fib/2).

fib(0,0). % fib(0)=0
fib(1,1). % fib(1)=1

fibM(N,Res) :- fib(N,Res),!.

fibM(N,Res) :-
    N>1,
    N1 is N-1,
    N2 is N-2,
    fibM(N1,R1),
    fibM(N2,R2),
    Res is R1+R2,
    asserta(fib(N,Res)).
```

Teste die folgenden Anfragen:

```
?- fib(40,R).
?- fib(100,R).
?- fib(1000,R).
```

Durch Memoisierung können rekursive Programme effektiver werden.

► Übung

Die Manipulation der Wissensbasis

- ist eine sehr mächtige Programmieretechnik. Aber, sie
- kann die Deklarativität eines Programms zerstören und
- kann zu sehr schwer lesbarem Code führen.
- Sie sollte mit Bedacht eingesetzt werden.

Sammlung aller Lösungen: findall/3

Das Prädikat `findall(O,Z,L)` generiert eine Liste L aller Objekte O, die das Ziel Z wahr machen.

- Gibt es kein solches Objekt so wird L mit der leeren Liste unifiziert.

```
figur(garfield, katze).
figur(snoopy, hund).
figur(popeye, mensch).
figur(garfield,7).
figur(snoopy, 7).
figur(popeye,30).
```

```
?- findall(Y,figur(X,Y),L).
L = [katze,hund,mensch,7,7,30].
```

Alle Lösungen einer Prologanfrage

- Auf Anfragen mit nicht instantiierten Variablen antwortet Prolog mit einer Variablenbelegung, die die Aussage wahr macht.
- Durch die Eingabe des Semikolons wird Prolog aufgefordert alternative Belegungen zu erzeugen.
- So können alle möglichen Lösungen nacheinander generiert werden.

```
food(sushi).
food(apple).
food(pudding).
mag(popeye,X):- food(X).

?- mag(popeye,X).
X=sushi;
X=apple;
X=pudding;
false.
```

- Mithilfe der eingebauten Prädikate `findall/3`, `bagof/3` und `setof/3` können alle Lösungen auf einmal generiert und in einer Liste gesammelt werden.

Sammlung aller Lösungen: findall/3

```
?- findall(name(X),figur(X,Y),L).
L = [name(garfield), name(snoopy), name(popeye),
     name(garfield), name(snoopy), name(popeye)].

?- findall(Y,(figur(X,Y),number(Y)),L).
L = [7,7,30].

?- findall((X,Y),figur(X,Y),L).
L = [(garfield, katze), (snoopy, hund), (popeye, mensch)
     (garfield, 7), (snoopy, 7), (popeye, 30)].

?- findall(Y,figur(snoopy,Y),L).
L = [hund, 7].

?- findall(Y,figur(hans,Y),L).
L = [].
```

Sammlung aller Lösungen: bagof/3

Das Prädikat `bagof(O,Z,L)` generiert eine Liste L aller Objekte O, die das Ziel Z wahr machen.

- Gibt es kein solches Objekt so scheidert das Prädikat.
- Zusätzlich werden freie Variablen gebunden.

```
?- bagof(Y,figur(X,Y),L).
X = garfield,
L = [katze, 7] ;
X = popeye,
L = [mensch, 30] ;
X = snoopy,
L = [hund, 7];
false.
```

Sammlung aller Lösungen: setof/3

Das Prädikat `setof(O,Z,L)` generiert eine Liste L aller Objekte O, die das Ziel Z wahr machen.

- Gibt es kein solches Objekt so scheidert das Prädikat.
- Zusätzlich werden freie Variablen gebunden.
- Doppelte Elemente in L werden entfernt.
- Die Elemente in L werden sortiert.

```
?- setof(X, Y^figur(X,Y),L).
[garfield,snoopy,popeye].
```

Sammlung aller Lösungen: bagof/3

```
27 ?- bagof(Y,figur(snoopy,Y),L).
L = [hund, 7].
```

```
?- bagof(Y,figur(hans,Y),L).
false.
```

```
?- bagof((X,Y),figur(X,Y),L).
L = [(garfield, katze), (snoopy, hund), (popeye, mensch)
, (garfield, 7), (snoopy, 7), (popeye, 30)].
```

```
?- bagof(Y,X^figur(X,Y),L).
L = [katze, hund, mensch, 7, 7, 30].
```

```
?- findall(L, bagof(Y,figur(X,Y),L),List).
List = [[katze, 7], [mensch, 30], [hund, 7]].
```

```
?- findall(f(X,L), bagof(Y,figur(X,Y),L),List).
List = [f(garfield, [katze, 7]), f(popeye, [mensch, 30]),
f(snoopy, [hund, 7])].
```

Sammlung aller Lösungen: setof/3

```
?- setof(Y,figur(X,Y),L).
X = garfield,
L = [katze, 7] ;
X = popeye,
L = [mensch, 30] ;
X = snoopy,
L = [hund, 7];
false.
```

```
?- setof(alter(Y), (figur(X,Y),number(Y)),L).
[alter(7),alter(30)].
```

```
?- setof(Y, figur(hans,Y), L).
false.
```

Dateiausgabe mit open/3

- In Prolog können Ausgaben (Streams) mit dem eingebauten Prädikat `open/3` in Dateien umgeleitet werden.
- Das Prädikat nimmt als erstes Argument einen Dateinamen. Das zweite Argument bestimmt den Modus (read, write, execute, default, all). Zuletzt wird eine Variable als Stream instanziiert.

```
write_file :- open('test.txt',write,FStream),
             write(FStream,'hello'),
             nl(FStream),
             close(FStream).

?- write_file.
```

Dateien einlesen mit consult/1

- Dateien können in Prolog mit dem Prädikat `consult/1` eingelesen werden.
- **Wichtig:** Die einzulesende Datei muss sich in Prolog-Notation befinden!
- Daten können bspw. in Form von Prädikaten eingelesen werden.

```
/* file.pl */
data(1,popeye).
data(1,pluto).
data(2,goofy).
data(2,mia).
data(1,anna).
```

```
load_file :- consult('file.pl').

?- load_file, findall(T,data(1,T),L).
L = [popeye, pluto, anna].
```

Prolog-Programme ausführen (aus Python)

- Prolog-Programme können aus anderen Programmiersprachen aufgerufen werden.
- Das nachfolgende Beispiel beschreibt einen möglichen Prolog-Aufruf aus Python heraus.

```
import subprocess

pl_file = "path/prolog_file.pl"

p = subprocess.Popen(["swipl","-q","-s",pl_file,"-g","main, halt", \
"-t","nl, halt"],stdout=subprocess.PIPE)

p.wait()
stdout,stderr = p.communicate()
```

Module

Module

- ermöglichen eine strukturierte Programmierung,
- verstecken Hilfsprädikate,
- erleichtern das Programmieren in Teams.

Arbeit mit Modulen

Deklarieren von Modulen:

```
% Datei tools.pl
% Modulname: tools
% oeffentliche Praedikate: reverse/2, member/2
:- module(reverse, [reverse/2, member/2]).
reverse(Liste,R) :- reverse_acc(Liste,[],R).
reverse_acc([],Acc,Acc).
reverse_acc([H|T],Acc,R) :- reverse_acc(T,[H|Acc],R).

member(X,[X|_]).
member(X,[_|_]) :- member(X,_).
```

Aufruf von Modulen

```
% Aus dem Modul reverse werden alle oeffentlichen Praedikate importiert
:- use_module(reverse).
```

```
% Aus dem Modul reverse wird das Praedikat reverse/2 importiert
:- use_module(reverse, [reverse/2]).
```

Graphviz: dot-Format

Graphviz ist eine Software zur Visualisierung von Graphen (<http://www.graphviz.org/>).

Zur Beschreibung von gerichteten Graphen wird das dot-Format verwendet:

```
digraph G {
main -> parse;
main -> init;
main -> cleanup;
parse -> execute;
execute -> make_string;
execute -> printf;
init -> make_string;
main -> printf;
execute -> compare;
}
```

Aufruf von der Shell:
\$ dot graph.dot -Tjpg -o graph.jpg

Libraries

- Libraries sind Module, die wichtige Prädikate zusammenfassen.
- Im SWI-Manual finden sie eine Liste der wichtigsten Libraries.

```
:- use_module(library(lists)).
```

Grafische Ausgabe eines Ableitungsbaums

Ziel: Erzeuge zu einem Baum in der Prolog-Term-Notation eine Beschreibung des Baums im dot-Format und generiere daraus ein Bild des Baums.

```
tree(1,s(np(det(die), n(katze)), vp(v(klaut), np(det(die), n(maus)))))).
```

Idee:

- Generiere für jeden Knoten eine dot-Knotendefinition:

```
q1[label=np];
```

- Generiere für jede Mutter-Tochter-Beziehung eine dot-Kante:

```
q2->q3;
```

Graphviz/DOT-Darstellung eines Ableitungsbaums (1)

```
:- dynamic(node/1).

% Graphviz/DOT (graphviz liegt im Unterordner gvdot)
dotjpeg :- shell('gvdot/dot.exe output.dot -Tjpg -o graph.jpg').

% Hauptaufruf
mgraph(N) :- open('output.dot',write,FS), tree(N,T), pdot(T,FS),
            close(FS), dotjpeg.

% Initialisierung des Tab - Zaehlers .
pdot(Term,FS):-
    retractall(node(_)),
    write(FS,'digraph G {'),nl(FS),
    assert(node(0)),pdot(0,Term,FS),
    write(FS,}')'.

% Beispielbaeume
tree(1,s(np(det(die), n(katze)), vp(v(klaut), np(det(die), n(maus))))).
tree(2,np(det(die), n(katze))).
```

Graphviz/DOT-Darstellung eines Ableitungsbaums (2)

```
% Baum drucken .
pdot(N,Term,FS):-
    Term =.. [F| Args ], % Struktur zu Liste .
    tab(FS,4),write(FS,q), write(FS,N), write(FS,'[label=')',
    write(FS,F),write(FS,']');', nl(FS), % Ausgabe des Mutterknotens .
    N1 is N+1, retract(node(_)), assert(node(N1)),
    pdot1(N,Args,FS). % Unterbaeume drucken .

% Unterbaeume drucken .
pdot1(Nmother,[H|T],FS):-
    node(Nchild), pdot(Nchild,H,FS), % Drucke eine Schwester .
    tab(FS,4),write(FS,q),write(FS,Nmother),write(FS,'->'),
    write(FS,q),write(FS,Nchild),write(FS,']');',nl(FS),
    pdot1(Nmother,T,FS). % Drucke die anderen Schwestern .
pdot1(_,[],_FS). % Termination .
```

Graphviz/DOT-Darstellung eines Ableitungsbaums (3)

```
?- mgraph(1).
```

generierte dot-Datei graph.dot

```
digraph G {
    q0[label=s];
    q1[label=np];
    q2[label=det];
    q3[label=die];
    q2->q3;
    q1->q2;
    q4[label=n];
    q5[label=katze];
    q4->q5;
    q1->q4;
    q0->q1;
    q6[label=vp];
    q7[label=v];
    [...]
}
```

generierte jpg-Datei graph.jpg

Zusammenfassung Kapitel 11 und 12

- Wir haben gelernt, wie wir dynamisch die Wissensbasis verändern können.
- Wir haben wichtige Prädikate zur Aufsammlung aller Lösungen einer Anfrage kennengelernt.
- Wir können Ergebnisse in Dateien schreiben und Dateien mit Prologklauseln einlesen.
- Wir haben gesehen, wie wir ein Prologprogramm in Module zerlegen können.

Übung: Memorisierung

In der letzten Sitzung haben wir ein Prädikat für die Fakultätsfunktion definiert:

```
fak(N,R):-  
    fak(N,1,R).  
  
fak(0,Acc,Acc):-!.  
  
fak(N,Acc,R):-  
    AccNew is N * Acc,  
    NNew is N - 1,  
    fak(NNew,AccNew,R).
```

Machen sie sich klar, dass die Fakultätsfunktion durch $fak(0) = 1$ und $fak(n) = fak(n-1) * n$ definiert werden kann.

Nutzen sie die Technik der Memoisierung für ein neues Prädikat zur Berechnung der Fakultät.

Warum führt die Memoisierung hier nicht zu derselben Effizienzsteigerung wie bei der Fibonacci-Folge? [▶ zurück](#)

Übung: alle Lösungen

Gegeben die folgende Datenbasis

```
q(blob,blug).  
q(blob,blag).  
q(blob,blig).  
q(blaf,blag).  
q(dang,dong).  
q(dang,blug).  
q(flaf,blob).
```

Was antwortet Prolog auf die folgenden Anfragen?

```
findall(X,q(blob,X),List).  
findall(X,q(X,blug),List).  
findall(X,q(X,Y),List).  
bagof(X,q(X,Y),List).  
setof(X,Y^q(X,Y),List).
```

Übung: assert, retract

Wie verändert sich die Datenbasis mit der folgenden Sequenz von Anfragen?

```
?- assert(q(a,b)), assertz(q(1,2)), asserta(q(foo,blug)).  
?- listing(q/2).
```

```
?- retract(q(1,2)), assertz( (p(X) :- h(X)) ).  
?- listing(q/2).
```

```
?- retractall(q(_,_)).  
?- listing(q/2).
```

Übung: max/2 mit Manipulation der Datenbasis

Schreiben Sie ein Prädikat `max/2`, das als erstes Argument eine Liste nimmt und dessen zweites Argument das Maximum aller Listenelemente ist.

Statt eines Akkumulators verwenden Sie `assert/1` und `retract/1` um Zwischenergebnisse zu speichern.

Bearbeiten Sie die practical session zu Kapitel 11 aus Learn Prolog Now.

Nehmen Sie ihr großes Grammatikprojekt und

- 1 lagern sie das Lexikon und die Grammatik in getrennte Module aus,
- 2 schreiben sie die generierten Ableitungsbäume in eine Datei,
- 3 schreiben sie ihre Testsätze in eine Datei, die sie in ihrem Grammatikprogramm einlesen,
- 4 schreiben sie ein Prädikat, das alle Testsätze nacheinander an den Parser übergibt und die Ableitungsbäume in eine Datei schreibt.

```
% tests.pl
ex(1,[die, katze, jagt, die, maus].
ex(2,[die, kleine, katze, klaut, hunde].
))
```