

Prolog

10. Kapitel: Cut und Negation

Dozentin: Wiebke Petersen

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

Das Prädikat `fail/0`

- Das Prädikat `fail/0` scheitert immer.
- Es erzwingt Backtracking und kann zur Ausgabe aller Lösungen eingesetzt werden:

```
all(L):-
    member(X,L),
    write(X),nl,
    fail.
all(_).
```

```
?- all([a,b,c]).
a
b
c
true.
```

Was ist die Aufgabe der zweiten `all`-Klausel?

Zusammenfassung Kapitel 9

- Wir haben verschiedene Prädikate zur Analyse von zusammengesetzten Termen kennengelernt:
 - `functor/3`
 - `arg/3`
 - `=.. /2` (univ)
- Wir haben gesehen, wie wir verschiedene Ausgaben auf dem Bildschirm erzeugen können und damit ein Prädikat `pprint/1` zur Ausgabe von Bäumen definiert.
 - `write_canonical/1` und `write/1`
 - `nl/0` und `tab/1`
- Wir haben gesehen, wie Operatoren definiert werden und die zentralen Eigenschaften von Operatoren kennengelernt:
 - Typ
 - Präzedenz
 - Assoziativität
- **Keywords:** `functor/3`, `arg/3`, `=.. /2`, `pprint/1`, Operatoren
- **Ausblick Kapitel 10:** Cut und Negation

Der Cut

Der Cut „!“ ist ein eingebautes Prädikat, mit dem Backtracking kontrolliert werden kann.

Der Cut kann folgendes bewirken:

- Effizienzsteigerung
- Speichereinsparung
- Kürzere Programme

Wirkungsweise:

- Der Cut wird im Rumpf von Regeln eingesetzt und verhindert Backtracking.
- Der Top-Down-Beweis des Cut gelingt immer.
- Nach dem Passieren eines Cuts in einem Regelrumpf sind
 - die Teilziele, die in demselben Regelrumpf vor dem Cut stehen, und
 - alle weiteren Klauseln desselben Prädikats, die hinter der Regel stehen,
 vom weiteren Backtracking ausgeschlossen.

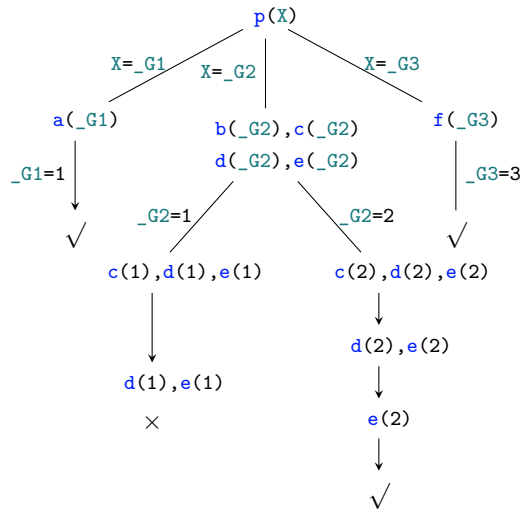
```

p(X) :- a(X).
p(X) :-
    b(X),
    c(X),
    d(X),
    e(X).
p(X) :- f(X).

a(1).
b(1).
b(2).
c(1).
c(2).
d(2).
e(2).
f(3).
    
```

```

?-p(X).
X=1;
X=2;
X=3;
false.
    
```



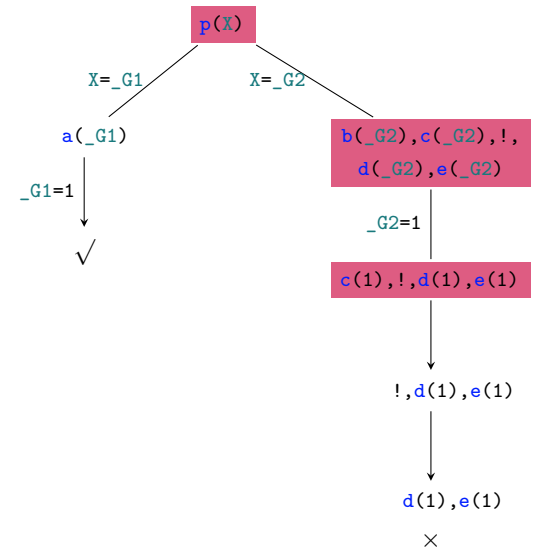
```

p(X) :- a(X).
p(X) :-
    b(X),
    c(X),!,
    d(X),
    e(X).
p(X) :- f(X).

a(1).
b(1).
b(2).
c(1).
c(2).
d(2).
e(2).
f(3).
    
```

```

?-p(X).
X=1;
false.
    
```



```

s(X,Y) :-
    q(X,Y).
s(0,0).

q(X,Y) :-
    i(X),
    j(Y).

i(1).
i(2).
j(1).
j(2).
j(3).
    
```

```

?- s(X,Y).
X = 1,
Y = 1;
X = 1,
Y = 2;
X = 1,
Y = 3;
X = 2,
Y = 1;
X = 2,
Y = 2;
X = 2,
Y = 3;
X = 0,
Y = 0;
false.
    
```

```

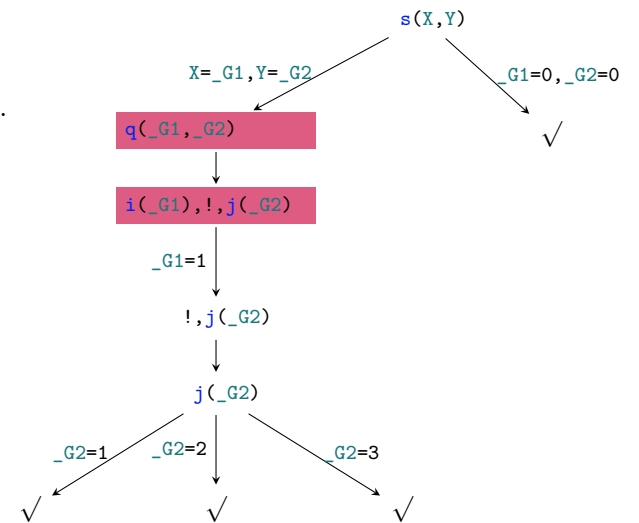
s(X,Y) :-
    q(X,Y).
s(0,0).

q(X,Y) :-
    i(X),!,
    j(Y).

i(1).
i(2).
j(1).
j(2).
j(3).
    
```

```

?-s(X,Y).
X=1,
Y=1;
X=1,
Y=2;
X=1,
Y=3;
X=0,
Y=0;
false.
    
```



In der Literatur wird oft zwischen roten und grünen Cuts unterschieden.

- Ein **grüner Cut** kann aus einem Programm entfernt werden, ohne dass sich die Bedeutung des Programms ändert.
- Ein **roter Cut** kann **nicht** aus einem Programm entfernt werden, ohne dass sich die Bedeutung des Programms ändert.

Das Prädikat `max(X,Y,Z)` soll gelingen, wenn Z das Maximum von X und Y ist.

ohne Cut:

```
max(X,Y,Y) :- X <= Y.
max(X,Y,X) :- X > Y.
```

mit grünem Cut:

```
max(X,Y,Y) :- X <= Y,!.
max(X,Y,X) :- X > Y.
```

mit rotem Cut:

```
max(X,Y,Y) :- X <= Y,!.
max(X,Y,X).
```

ineffizient!

```
?- max(3,5,X).
1 Call: max(3,5,_487) ?
2 Call: 3=<5 ?
2 Exit: 3=<5 ?
1 Exit: max(3,5,5) ?
X = 5 ? ;
1 Redo: max(3,5,5) ?
2 Call: 3>5 ?
2 Fail: 3>5 ?
1 Fail: max(3,5,_487) ?
false.
```

gut!

```
?- max(3,5,X).
1 Call: max(3,5,_487) ?
2 Call: 3=<5 ?
2 Exit: 3=<5 ?
1 Exit: max(3,5,5) ?
X = 5 ? ;
false.
```

falsch!

```
?- max(3,5,3).
true.
```

► Übung

Schattenseiten des Cuts

- Der Cut zerstört die Deklarativität von Prolog-Programmen.
- Die Interpretation einer Prädikatsdefinition mit roten Cuts ist i.d.R. nur noch unter Berücksichtigung der Reihenfolge der Beweisschritte möglich.
- Deshalb: Cut nur einsetzen, wenn ein offensichtlicher Vorteil erzielt werden kann.

Gründe für die Verwendung des Cuts

- Beschneiden des Suchraums.
- Erzwingen von Determinismus.
- Modellierung von Defaults.
- Modellierung von Negation.

```
my_member(H,[H|_]) :- !.
my_member(H,[_|T]) :- my_member(H,T).
```

```
?- my_member(b,[a,b,c]).
true.
?- my_member(d,[a,b,c]).
false.
?- my_member(X,[a,b,c]).
X=a.
true.
?- my_member(a,X).
X=[a|G_1].
true.
```

Die Beschneidung des Suchraums führt zu Determinismus.

L2 entsteht durch das Löschen eines Vorkommens von X aus L1.

```
% delete_once(X,L1,L2)
delete_once(X,[X|L1],L1).
delete_once(X,[H|T1],[H|T2]):-
    delete_once(X,T1,T2).
```

```
?- delete_once(a,[a,b,c,a,d],L).
L = [b, c, a, d] ;
L = [a, b, c, d] ;
false.
```

L2 entsteht durch das Löschen des ersten Vorkommens von X aus L1.

```
% delete_first(X,L1,L2).
delete_first(X,[X|L1],L1):-!.
delete_first(X,[H|T1],[H|T2]):-
    delete_first(X,T1,T2).
```

```
?- delete_first(a,[a,b,c,a,d],L).
L = [b, c, a, d].
```

$$\text{Zur Erinnerung: } n! = \underbrace{1 \cdot 2 \cdot \dots \cdot n}_{n\text{-mal}}$$

Lösung nicht deterministisch:

```
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).
```

Problem: Prädikat im Backtracking ⇒ Endlosschleife

Verbesserung durch Kontrollabfrage

```
fak(N,R):-
    fak(N,1,R).

fak(0,Acc,Acc).

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

```
?- fak(5,X).
X=120;
false.
```

Determinismus durch Cut:

```
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).
```

```
?- fak(5,X).
X=120.
```

```
1 neg(A):-
2   A,
3   !,fail.
4 neg(_).
```

```
hund(snoopy).
hund(pluto).
katze(garfield).

?- neg(katze(pluto)).
true.
?- \+ katze(pluto).
true.

?- neg(katze(X)).
false.
?- \+ katze(X).
false.
```

Negation wird in Prolog durch eine Cut-Fail-Kombination realisiert („negation as failure“).

- Z. 2: Wenn der Ausdruck **A** bewiesen werden kann,
- Z. 3: sorgt die Cut-Fail-Kombination dafür, dass der Beweis von **neg(A)** scheitert. Der Cut hinter **A** und vor **fail** verhindert, dass die zweite Klausel von **neg(A)** für eine beweisbare Aussage **A** herangezogen werden kann.
- Z. 4: Greift die erste Klausel nicht, ist **A** nicht beweisbar und die Negation von **A** ist wahr.

Das Negationsprädikat ist in Prolog als Präfix-Operator **\+** vordefiniert.

Vorsicht: Enthält eine Aussage Variablen und gibt es eine Variablenbelegung, die die Aussage wahr macht, ist die Negation der Aussage falsch.

```
1 ifthenelse(If,Then,_):-
2   If,
3   Then.
4 ifthenelse(If,_ ,Else):-
5   \+ If,
6   Else.
```

- Drei Aussagen **If**, **Then** und **Else** erfüllen die if-then-else-Relation, wenn
- Z. 6-8: entweder **If** und **Then** beweisbar sind oder
- Z. 9-11: **If** nicht beweisbar und **Else** beweisbar ist.

Beispiel: max/3 mit ifthenelse/3:

```
max(X,Y,Z):-
    ifthenelse(X<=Y,Y=Z,X=Z).
```

Wenn **X** kleiner oder gleich **Y** ist, so ist **Y** das Maximum von **X** und **Y**.

Wenn **X** nicht kleiner oder gleich **Y** ist, so ist **X** das Maximum von **X** und **Y**.

```

1 neg(A):-
2   A,
3   !,fail.
4 neg(_).
5
6 ledigerStudent(X):-
7   neg(verheiratet(X)),
8   student(X).
9
10 student(peter).
11 verheiratet(klaus).

```

Z. 6-8: X ist lediger Student, wenn X nicht verheiratet ist und wenn X Student ist.

- „Negation as failure“ ist keine logische Negation. Daher kann das Prädikat `ledigerStudent/1` in dieser Form nicht zur Generierung aller ledigen Studenten eingesetzt werden.
- Sehen Sie einen einfachen Weg, das Prädikat zu verbessern?

```

?- ledigerStudent(peter).
true.
?- ledigerStudent(klaus).
false.
?- ledigerStudent(X).
false.

```

mit cut-fail direkt

```

can_fly(X):-
  penguin(X),
  !, fail.
can_fly(X):- bird(X).

bird(X):- penguin(X).
bird(X):- eagle(X).

penguin(tweety).
eagle(klaus).

```

schlecht!

```

?- can_fly(tweety).
false.
?- can_fly(klaus).
true.
?- can_fly(popeye).
false.
?- can_fly(X).
false.

```

mit Negation (Version 1)

```

can_fly(X):-
  neg(penguin(X)),
  bird(X).

bird(X):- penguin(X).
bird(X):- eagle(X).

penguin(tweety).
eagle(klaus).

```

ebenso schlecht!

```

?- can_fly(tweety).
false.
?- can_fly(klaus).
true.
?- can_fly(popeye).
false.
?- can_fly(X).
false.

```

mit Negation (Version 2)

```

can_fly(X):-
  bird(X),
  neg(penguin(X)).

bird(X):- penguin(X).
bird(X):- eagle(X).

penguin(tweety).
eagle(klaus).

```

gut!

```

?- can_fly(tweety).
false.
?- can_fly(klaus).
true.
?- can_fly(popeye).
false.
?- can_fly(X).
X=klaus.
true.

```

- 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.

- 1 Nehmen sie ihr Grammatikprogramm und erweitern sie es um das Prädikat `generate_all/1`:
 - `generate_all/1` nimmt eine natürliche Zahl als Argument (Bsp.: `generate_all(5)`) und
 - generiert alle grammatischen Sätze dieser Länge.
 - Die Sätze werden an ihr Prädikat `parse/1` weitergereicht, so dass alle Sätze sowie alle Ableitungsbäume auf dem Bildschirm ausgegeben werden.

Tipp: Verwenden sie `fail/0` um Backtracking zu erzwingen.

- 2 Schreiben sie ein zweistelliges Prädikat `generate_all/2`, das zwei Zahlen als Argument nimmt (Bsp. `generate_all(5,7)`) und alle Sätze generiert, deren Länge im Intervall der beiden Zahlen liegt.
- 3 Schreiben sie durchnummerierte Beispielsätze in ihre Wissensbasis (`ex/2`, siehe Übung zu Kapitel 9). Schreiben sie ein Prädikat `parse_all/0`, das alle Beispielsätze der Wissensbasis parst.

Tipp: Verkleinern sie in der Entwicklungsphase ihr Lexikon, um die Ergebnismenge kleiner zu halten.

Gegeben folgende Wissensbasis:

```
p(1).
p(2) :- !.
p(3).
```

Was antwortet Prolog auf folgende Anfragen?

- 1 ?- p(X).
- 2 ?- p(X), p(Y).
- 3 ?- p(X), !, p(Y).

▸ zurück

Handelt es sich bei folgendem Cut um einen roten oder grünen?

```
append([], L, L) :- !.
append([_ | T1], L, [_ | T2]) :- append(T1, L, T2).
```

Tipp: Was passiert bei der Anfrage ?- append(X, Y, [a, b, c]).?

▸ zurück

Was macht das folgende Prädikat?

```
class(Number, positive) :- Number > 0.
class(0, zero).
class(Number, negative) :- Number < 0.
```

Können Sie es durch den Einsatz eines grünen Cuts verbessern?

▸ zurück

- 1 Wenn sie noch Zeit haben, bearbeiten sie bitte auch die Aufgaben 10.3-10.5 aus den Übungen zu Kapitel 10 in Learn Prolog Now.
- 2 Schauen sie sich außerdem die "Practical Session" zu Kapitel 10 in Learn Prolog Now an.