Logische und funktionale Programmierung

Vorlesung 5&6&7: Listen und Backtracking

Babeş-Bolyai Universität, Department für Informatik, Cluj-Napoca
csacarea@cs.ubbcluj.ro

Rekursive Suche in Listen

Wir suchen in einer Liste nach Information.

Liste von Städten:

```
[shanghai, manchester, vancouver, portland]
```

Aufgabe: ist eine bestimmte Stadt in der Liste?

1. `head` ist die Stadt
2. `stadt` ist in tail $\rightarrow$
   1. head of tail ist die Stadt
   2. stadt ist in tail of tail
      1. head of tail of tail ist die stadt
      2. stadt ist in tail of tail of tail

Rekursive Suche in Listen

Prologimplementation:

`member(X,Y).` `true` wenn `X` in der Liste ist , die durch `Y` repräsentiert ist.

1. `member(X,[Y|_]) :- X=Y.`
2. `member(X,[_|Y]) :- member(X,Y).`

1. als Grenzbedingung,
2. als Rekursionsfall.

Rekursionsbewegung auf ein Ende zu: jeder Neuaufruf bekommt eine kürzere Liste.

Alternative Schreibweise:

(1) `member(X,[X|_].`

Rekursion Fallen

Vorsicht vor zirkulären Definitionen

```
parent(X,Y) :- child(Y,X).
child(A,B) :- parent(B,A).
```

Anfrage `parent` oder `child` führt zu niemals endender Suche.

Vorsicht Linksrekursion:

Eine Regel verursacht ein neues Ziel welches mit demjenigen äquivalent ist, das dazu führte, dass die Regel aufgerufen wurde.

```
person(X):-person(Y), mother(X,Y).
person(adam).
?- person(X).% erste Regel wird ewig
```

Rekursion Fallen

Die bloße Angabe der relevanten Fakten und Regeln stellt nicht sicher, dass Prolog sie immer finden wird: wie sucht Prolog, welche Variablen werden wie instantiert wenn eine der Regeln benutzt wird.

Generell:

Fakten vor Regeln, wann immer möglich.

Rekursion Fallen

```
islist([A|B]) :- islist(B).
islist([ ]).
?- islist([ a,b,c,d]).
?- islist([ ]).
?- islist(f(1,2,3)).
?- islist(X).
```

Wie kann man das Prädikat ändern, so dass es keine unendliche Schleife erzeugt?

Prädikat Schwache Liste

```
weak_islist([ ]).
weak_islist([_ | _]).
```

Rekursiver Vergleich

Prolog besitzt eingebaute Prädikate um Integer zu vergleichen. Ein Strukturvergleich ist komplizierter. Es müssen alle Komponenten verglichen werden. Sind Komponenten Strukturen, muss der Vergleich rekursiv sein.

Beispiel Verbrauchsstatistik

Fahre verschiedene Autos auf verschiedenen Routen und messe den Benzinverbrauch

```
fuel_consumed(suv,[13.1,10.4,...]).
fuel_consumed(sedan,...).
fuel_consumed(hybrid,...).
```

Rekursiver Vergleich

Ein Autotyp soll gleich oder besser als ein anderer sein, wenn seine Verbrauchsmenge mindestens 5% besser ist als der Durchschnitt beider ist.
Entsprechend 1/40 der Summe beider Verbrauchswerte.

Informal:

```
equal_or_better_ongas(Good, Bad):-
Threshold is ((Good + Bad)/2*0.95), Good =<
Threshold.
```

Rekursiver Vergleich

Eine Liste ist immer besser als eine andere, wenn ihr Kopf `equal_or_better_ongas` erfüllt und ihr Schwanz `always_better` erfüllt.

- Prädikat geht beide Testlisten nach rechts bis sie am Ende sind.
- Dann muss ein Abbruchprädikat die Rekursion beenden.
- Wenn nur einmal `equal_or_better` scheitert, scheitert `always_better` und damit das Vergleichsprädikat `prefer`.

Rekursiver Vergleich

```
prefer(Auto1, Auto2):-
fuelconsumed(Auto1, Mengen1),
fuelconsumed(Auto2, Mengen2),
always_better(Mengen1,Mengen2).
always_better([ ],[ ]).
always_better([Strecke_a|Rest_a],
[Strecke_b|Rest_b]) :-
equal_or_better_ongas(Strecke_a, Strecke_b),
always_better(Rest_a,Rest_b).
```

Rekursiver Vergleich

Aufgabe:

implementieren Sie das Ganze mit einem Prädikat `sometimes_better` das schon matcht wenn einmal `equal_or_better_ongas` zutrifft.

Die Rekursion im Detail

Den Mechanismus der Rekusion zu verstehen bereitet 'Iterations-' Programmierer im allgemeinen große Schwierigkeiten.
Das Verfahren der Rekursion basiert auf der Möglichkeit, dass sich eine Funktion selbst aufrufen kann. Mit Hilfe dieser Möglichkeit können Programmteile wiederholt werden und somit kann eine Iterations-schleife simuliert werden.
Da in der Programmiersprache PROLOG keine Iterationsschleife als Ablaufsteuerungselement vorhanden ist, muss jegliche Wiederholungsschleife als rekursive Funktion definiert werden. Selbstverständlich darf nicht vergessen werden, dass die Schleife und die Rekursion zu einem Ende kommen muss, d.h. sie muss terminieren.

REKURSIONSBEISPIEL

PROLOG	PERL

drucke_bis_zehn(N):-
  N=<10,
  write(N),
  N1 is N + 1,
  drucke_bis_zehn(N1).
 	
Aufruf:
?- drucke_bis_zehn(1).
	sub drucke_bis_zehn()
	{
	my ($zahl);
	$zahl = @_[1]; #holt das Argument
	while ($zahl <= 10)
	 {
	  print($zahl\n);
	  $zahl++;
	 }
	}
	
	Aufruf:
	drucke_bis_zehn(1);

Die Rekursion im Detail

Etwas schwieriger wird es, wenn die Wiederholungsanweisung ein Ergebnis berechnen soll. Die Programmiersprache PERL hat die Möglichkeit über Variablen sich das Zwischenergebnis zu merken und am Ende der Wiederholung das Zwischenergebnis als Endergebnis zurückzugeben.
Da die Programmiersprache PROLOG bei jeder Wiederholung die selbe Funktion aufruft, die für sich in einem eigenen Kontext abläuft, d.h. eigene Variablen und Argumente hat, kann keine Variable die Werte zwischen den Aufrufen der Funktion (auch Instanzen der Funktion genannt) speichern. Der PROLOG Programmierer kann keine globalen Variablen verwenden.
Die große Frage lautet: Wie kann PROLOG sich von einer Instanz der Funktion zur nächsten Werte merken ? Hier hilft die Implementation des Mechanismus der Rekursion.

Die Rekursion im Detail

Bei jeder Instanz einer rekursiven Funtion merkt sich der Computer in seinem Speicher (genannt STACK) die aktuellen Werte der Variablen dieses Aufrufs. Die Werte jeder Variable innerhalb einer Instanz bleiben also im Stack gespeichert. Terminiert die Funktion, kann die Funktion mit Hilfe des Backtrackings auf die im STACK gespeicherten Variablen zurückgreifen.
Im nächsten Beispiel soll gezeigt werden, wie die PROLOG rekursiv und PERL iterativ die Zahlen zwischen Null und N addieren kann.

PROLOG vs. PERL

PROLOG

```
addiere_bis_N(N,0):-
  N=0.
addiere_bis_N(N,X):-
  N1 is N - 1,
  addiere_bis_N(N1,Y),
  X is Y + N.

Aufruf:
?- addiere_bis_N(2,N).
```

PERL iterativ

```
sub addiere_bis_N_iter()
{
 my $zahl = @_[0];
 my $erg = 0;

 while ($zahl>=0)
 {
    $erg = $erg + $zahl;
    $zahl--;
 }
 return $erg;

}
```

PERL rekursiv

```
sub addiere_bis_N()
{
 my $zahl = @_[0];   #Argument
 if ($zahl==0)
    {
 return 0;
    }
    else
    {
 $hilfszahl = $zahl - 1;
 $y = &addiere_bis_N($hilfszahl);
 $zahl = $y + $zahl;
 return $zahl;
    }
}
```

PROLOGPROGRAMM IN DETAIL

```
addiere_bis_N(N,0):-                    %Zeile 1
 N=0.                                   %Zeile 2
addiere_bis_N(N,X):-                    %Zeile 3
 N1 is N - 1,                           %Zeile 4
 addiere_bis_N(N1,Y),                   %Zeile 5
 X is Y + N.                            %Zeile 6
```

Aufruf

```
?- trace.
?- addiere_bis_N(2,N).


[trace] 21 ?- %Zeile 1
   Call: (7) addiere_bis_N(2, _G483) ? creep
   Call: (8) 2=0 ? creep
   Fail: (8) 2=0 ? creep
   Redo: (7) addiere_bis_N(2, _G483) ? creep
^  Call: (8) _L170 is 2-1 ? creep
^  Exit: (8) 1 is 2-1 ? creep
   Call: (8) addiere_bis_N(1, _L171) ? creep
   Call: (9) 1=0 ? creep
   Fail: (9) 1=0 ? creep
   Redo: (8) addiere_bis_N(1, _L171) ? creep
^  Call: (9) _L182 is 1-1 ? creep
^  Exit: (9) 0 is 1-1 ? creep
   Call: (9) addiere_bis_N(0, _L183) ? creep
   Call: (10) 0=0 ? creep
   Exit: (10) 0=0 ? creep
   Exit: (9) addiere_bis_N(0, 0) ? creep
^  Call: (9) _L171 is 0+1 ? creep
^  Exit: (9) 1 is 0+1 ? creep
   Exit: (8) addiere_bis_N(1, 1) ? creep
^  Call: (8) _G483 is 1+2 ? creep
^  Exit: (8) 3 is 1+2 ? creep
   Exit: (7) addiere_bis_N(2, 3) ? creep

X = 3
```

AUFRUF

erste Instanz: addiere_bis_N(2,N).

Das Adressbuch und ihre Werte innerhalb der ersten Instanz:

Argument/Variablen name	Instanz Name	Wert
N	N	2
X	_G483	noch nicht bekannt
N1	_L170	1
Y	_L171	noch nicht bekannt

Die erste Instanz läuft in der Zeile 5 auf einen erneuten Funktionsaufruf und erzwingt eine zweite Instanz. Falls die zweite Instanz abgearbeitet ist, dann wird an der Zeile 6 weitergerabeitet. (outstanding 1)

outstanding 1: Zeile 6: X is Y + N , X : _G483 is _L171 + 1

AUFRUF

zweite Instanz: addiere_bis_N(1,N).

Das Adressbuch und ihre Werte innerhalb der zweiten Instanz:

Argument/Variablen name	Instanz Name	Wert
N	N	1
X	_L171	noch nicht bekannt
N1	_L182	0
Y	_L183	noch nicht bekannt

Die zweite Instanz läuft in der Zeile 5 auf einen erneuten Funktionsaufruf und erzwingt eine dritte Instanz. Falls die dritte Instanz abgearbeitet ist, dann wird an der Zeile 6 weitergerabeitet. (outstanding 2)

outstanding 2: Zeile 6: X is Y + N , X : _L171 is _L183 + 1

AUFRUF

dritte Instanz: addiere_bis_N(0,N).

```
addiere_bis_N(N,0):-                                    %Zeile 1
  N=0.                                                  %Zeile 2
```

Das Adressbuch und ihre Werte innerhalb der dritten Instanz:

Argument/Variablen name	Instanz Name	Wert
N	N	1
X		0

In der dritten Instanz terminiert die Funktion und das zweite Argument bekommt den Wert 0.

AUFRUF

```
Jetzt wird das Backtracking gestartet und es können die ausstehenden
Operationen  ausgeführt werden:
outstanding 2
und
outstanding 1

X = 0, Y = _L183

outstanding 2:  Zeile 6: X is Y + N ,   X :  _L171 is _L183 + 1

Y = _L183 + 1
unifikation: X = Y,
X = 2

outstanding 1: Zeile 6: X is Y + N ,   X :  _G483 is _L171 + 2
```

AUFRUF

```
Y = _L171 + 2
unifikation: X = Y
X = 3
```

Der Aufruf tabellarisch:

erste Instanz	zweite Instanz	dritte Instanz	erste Inst.	zweite Instanz	dritte Instanz
addiere_bis_N(2,X):-			%Zeile 3		
N1 is 2 - 1,			%Zeile 4		
addiere_bis_N(1,Y),			%Zeile 5		
--------------→	addiere_bis_N(1,X):-			%Zeile 3	
	N1 is 1 - 1,			%Zeile 4	
	addiere_bis_N(0,Y),			%Zeile 5	
	--------------→	addiere_bis_N(0,0),			%Zeile 1
		X=0 %Terminierung			%Zeile 2
		←- Backtracking			
	X is 0 + 1.			%Zeile 6	
	←- Backtacking				
X is 2 + 1.			%Zeile 6		

JOIN VON STRUKTUREN

Dreistelliges Prädikat, das zwei Listen in einer neuen Liste zusammenhängt

```
?- append([a,b,c],[3,2,1],[a,b,c,3,2,1]).
?- append([alpha,beta],[gamma,delta],X).
X=[alpha,beta,gamma,delta]
append(X,[b,c,d],[a,b,c,d]).
X=a.
%was ist hier falsch?
```

Join von Strukturen

- Grenzbedingung: erste Liste ist leere Liste
- Jede Liste, die der leeren Liste angehängt wird ist diese Liste.

Rekursionsfall:

1. das erste Element der ersten Liste wird das erste Element der dritten Liste (Ausgabeliste).
2. dem Schwanz (Rest) der ersten Liste wird die zweite Liste angehängt um den Schwanz der dritten Liste zu bilden.
3. verwende `append` selbst um (2) durchzuführen.
4. da im Rekursionsfall das erste Argument ständig reduziert wird, kommt es zur Grenzbedingung.

JOIN VON STRUKTUREN

```
append([ ], L,L).
append([X|L1],L2[X|L3]):- append(L1,L2,L3).
```

Join von Strukturen

Aufruf von myappend mit:
[trace] 26 ?- myappend([1,2],[3,4],N).

Call: (7) myappend([1, 2], [3, 4],N=_G485) ? creep

entspricht:

```
myappend([X=1|L1=[2]],L2=[3,4],[1,G_554]) :-
    myappend([2],[3,4],_G554).
```

Call: (8) myappend([2], [3, 4],_G554) ? creep

entspricht:

```
myappend([X=2|L1=[]], L2=[3,4],[2,_G557]) :-
    myappend([],[3,4],_G557).
```

Call: (9) myappend([], [3, 4], _G557) ? creep

Join von Strukturen

Da das erste Argument die leere Liste ist, wird die Terminierungsregel aufgerufen und `_G557` bekommt den konkreten Startwert:

```
_G557=[3,4]
Exit: (9) myappend([], [3, 4], [3, 4]) ? creep
```

Um den gesuchten Wert von N zu erhalten müssen die Hilfsvariablen `_G485` und `_G554` zurückverfolgt werden um daraus N zu rekonstruieren.

```
_G554 = [2,_G557] = [2,3,4]
Exit: (8) myappend([2], [3, 4], [2, 3, 4]) ?
creep
_G485 = [1,_G554] = [1,2,3,4]
```

JOIN VON STRUKTUREN

```
Exit: (7) myappend([1, 2], [3, 4], [1, 2, 3,
4]) ? creep
```

und schließlich:

```
N = G_485
N = [1, 2, 3, 4]
```

Listenzerlegung mit append

Durch Einsatz von Variablen kann `append` auch in die andere Richtung zur Listenzerlegung benutzt werden.

```
?- append( L1,L2,[a,b,c]).
L1 = [ ] L2=[a,b,c];
L1 = [a] L2=[b,c];
L1 = [a,b] L2=[c];
L1 = [a,b,c ] L2=[];
no
```

Erzeugung einer bestimmten Liste mit append

Wir können nach einem bestimmten Pattern in einer Liste suchen.

```
?- append( Before,[may|After],
[jan,feb,mar,apr,may,jun.jul,aug,sep,oct,nov,dec]).
Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec]
```

Backtracking:

Suche nach einer Lösung

- Zur Erinnerung: Bislang haben wir Anfragen an die Wiba gestellt, die mit der Meldung IC-Verbindung existiert (im Fall der Ableitbarkeit des Goals) bzw. IC-Verbindung existiert nicht (im Fall der Nicht-Ableitbarkeit des Goals) beantwortet wurden.
- Bei der Ableitbarkeits-Prüfung setzte Backtracking einzig und allein dann ein, wenn es um den Nachweis ging, ob überhaupt eine Ableitung möglich ist.
- Bei einer erfolgreichen Ableitung wurde die Programmausführung beendet, ohne dass ein weiteres Backtracking erfolgte.

Backtracking:

Suche nach allen Lösungen

- Im folgenden werden wir darstellen, wie das Backtracking der Inferenzkomponente beeinflußt werden kann, um z.B. Fragestellungen nach sämtlichen Lösungsalternativen - als insgesamt aus der Wiba ableitbarem Wissen über IC-Verbindungen - bearbeiten lassen zu können.
- Dazu betrachten wir die folgende Aufgabenstellung:
- Durch die Angabe eines Abfahrtsorts sollen alle möglichen Ankunftsorte angezeigt werden, die über IC-Verbindungen erreicht werden können

Beispiel

- Anfrage nach IC-Verbindungen; Anforderung zur Eingabe von Abfahrts- und Ankunftsort über internes Goal mit dem Prädikat `anfrage`;

BEISPIEL

```
dic(ha,kö).
dic(ha,fu).
dic(kö,ka).
dic(kö,ma).
dic(fu,mü).
dic(ma,fr).

ic(Von,Nach):-dic(Von,Nach).
ic(Von,Nach):-dic(Von,Z),ic(Z,Nach).

anfrage:-write(' Gib Abfahrtsort: '),nl,
        ttyread(Von),
        write(' Gib Ankunftsort: '),nl,
        ttyread(Nach),
        ic(Von,Nach),
        write(' IC-Verbindung existiert ').
anfrage:-write(' IC-Verbindung existiert nicht ').

:- anfrage.
```

Erklärungen

- Das PROLOG-System fordert durch ?- immer ein externes Goal an, sofern es nicht durch den Eintrag eines internen Goals innerhalb des PROLOG-Programms daran gehindert wird.
- Wir legen ein internes Goal dadurch fest, dass wir die Zeichen :- ans Ende der Wiba und dahinter - mit abschlieendem Punkt . - geeignete Prädikate für eine Anfrage eintragen, die beim Programmstart als Goal aufgefaßt werden soll.

Erklärungen

- Das PROLOG-System fordert durch ?- immer ein externes Goal an, sofern es nicht durch den Eintrag eines internen Goals innerhalb des PROLOG-Programms daran gehindert wird.
- Wir legen ein internes Goal dadurch fest, dass wir die Zeichen :- ans Ende der Wiba und dahinter - mit abschlieendem Punkt . - geeignete Prädikate für eine Anfrage eintragen, die beim Programmstart als Goal aufgefaßt werden soll.
- :-anfrage.

Erklärungen

- Um einen Bezug zu unseren bisherigen Regeln herzustellen, muss das Prädikat `anfrage` der Kopf einer Regel sein, in deren Rumpf das Prädikat `ic` - zusammen mit weiteren Prädikaten - für den Dialog mit dem Anwender enthalten ist.
- Wichtig für den Dialog mit dem Anwender ist das Standard-Prädikat `write`, mit dem sich Text auf dem Bildschirm anzeigen läßt.
- Dieser Text muss als Argument des Prädikats `write` aufgeführt werden.
- Er wird in dem Augenblick ausgegeben, in dem das Prädikat `write` unifiziert wird.

ERKLÄRUNGEN

- Bei der Unifizierung des Prädikats `write( Gib Abfahrsort: )` wird der Text
  Gib Abfahrtsort:
  am Bildschirm angezeigt.

```
anfrage:-   write(' Gib Abfahrtsort: '),nl,
            ttyread(Von),
            write(' Gib Ankunftsort: '),nl,
            ttyread(Nach),
            ic(Von,Nach),
            write(' IC-Verbindung existiert ').
```

Erklärungen

- Bei der Unifizierung des Prädikats `ttyread` wird eine Eingabe von der Tastatur angefordert.
- Die als Argument aufgeführte Variable `Von` wird mit demjenigen Wert instanziert, der über die Tastatur eingegeben wird.
- Das Prädikat `nl` beeinflußt die Bildschirmausgabe.
- Die Unifizierung dieses Prädikats, das kein Argument besitzt, hat zur Folge, dass der Cursor auf den Anfang der nächsten Bildschirmzeile positioniert wird.

Backtracking

- Die Standard-Prädikate `write, ttyread` und `nl` von der Inferenzkomponente werde niemals als Backtracking-Klauseln markiert, da sie keine Alternativen für eine Unifizierung zulassen.
- Diese Prädikate sind nicht Backtracking fähig

Erweiterung 1

Wie werden mögliche Zwischenstationen gezeigt?

```
ic(Von,Nach):-dic(Von,Nach).
ic(Von,Nach):-dic(Von,Z),
write(mögliche Zwischenstation: ),
write(Z),nl, ic(Z,Nach).
```

Erweiterung 2

- Wie können wir neben den möglichen Zwischenstationen zusätzlich die Nummern der unifizierten Fakten (von 1 bis 6) sowie die Nummern der unifizierten Regelköpfe (1 oder 2) anzeigen lassen?

Erweiterung 2

- Wie können wir neben den möglichen Zwischenstationen zusätzlich die Nummern der unifizierten Fakten (von 1 bis 6) sowie die Nummern der unifizierten Regelköpfe (1 oder 2) anzeigen lassen?
- Wir verändern die Prädikate `dic` und `ic` indem wir jeweils ein zusätzliches Argument mit der zugehörigen Positionsnummer als 1. Argument einfügen.
- Zusätzlich tragen wir in die Regeln mit dem Prädikatsnamen `ic` die Prädikate `write` und `nl` zur Ausgabe der jeweiligen Regel-Nummer an den Anfang des Regelrumpfes ein.

ERWEITERUNG 2

```
dic(1,ha,kö).
dic(2,ha,fu).
dic(3,kö,ka).
dic(4,kö,ma).
dic(5,fu,mü).
dic(6,ma,fr).

ic(1,Von,Nach):-write(' Regel: 1 '),nl,
        dic(Nr,Von,Nach),
        write(' Fakt: '),write(Nr),nl.
ic(2,Von,Nach):-write(' Regel: 2 '),nl,
        dic(Nr,Von,Z),
        write(' Fakt: '),write(Nr),nl,
        write(' mögliche Zwischenstation: '),write(Z),nl,
        ic( _ ,Z,Nach).
```

Erweiterung 3

- Anforderung zur Eingabe von Abfahrts- und Ankunftsort über internes Goal mit dem Prädikat `anfrage`;
- Ausgabe der Fakten-Nummern und der Regel-Nummern, sowie der möglichen Zwischenstationen als Erklärung fur die Arbeit der Inferenzkomponente;

ERWEITERUNG 3

```
dic(1,ha,kö).
dic(2,ha,fu).
dic(3,kö,ka).
dic(4,kö,ma).
dic(5,fu,mü).
dic(6,ma,fr).

ic(1,Von,Nach):-write(' Regel: 1 '),nl,
        dic(Nr,Von,Nach),
        write(' Fakt: '),write(Nr),nl.
ic(2,Von,Nach):-write(' Regel: 2 '),nl,
        dic(Nr,Von,Z),
        write(' Fakt: '),write(Nr),nl,
        write(' mögliche Zwischenstation: '),write(Z),nl,
        ic(_,Z,Nach).

anfrage:-write(' Gib Abfahrtsort: '),nl,
        ttyread(Von),
        write(' Gib Ankunftsort: '),nl,
        ttyread(Nach),
        ic(_,Von,Nach),
        write(' IC-Verbindung existiert ').
anfrage:-write(' IC-Verbindung existiert nicht ').

:- anfrage.
```

Suche nach allen Lösungen

- Durch die Angabe eines Abfahrtsorts sollen alle möglichen Ankunftsorte angezeigt werden, die über IC-Verbindungen erreicht werden können
- Die zugehörige Lösung entwickeln wir aus dem Programm mit dem wir die Existenz einer IC-Verbindung zwischen einem Abfahrts- und einem Ankunftsort prüfen konnten.
- Dazu sind Änderungen bei der Formulierung des internen Goals und der Regeln mit dem Regelkopf `anfrage` erforderlich.

```
anfrage:-write(' Gib Abfahrtsort: '),nl,
        ttyread(Von),
        write(' Gib Ankunftsort: '),nl,
        ttyread(Nach),
        ic(Von,Nach),
        write(' IC-Verbindung existiert '),nl.
anfrage:-write(' IC-Verbindung existiert nicht '),nl.
```

- Kein Ankunftsort $\longrightarrow$ löschen des 4., 5. und 6. Prädikats.
- Was passiert jetzt?

- Kein Ankunftsort $\longrightarrow$ löschen des 4., 5. und 6. Prädikats.
- Was passiert jetzt? Variable `Nach` wird jetzt nicht mehr instanziert.
- Das Subgoal `ic(Von,Nach)` wird immer ableitbar ist, wenn es eine weitere Instanzierung für die Variable `Nach` gibt.
- Damit derartige Instanzierungen - bei einem internen Goal - angezeigt werden, können wir das Standard-Prädikat `write` mit der Variablen `Nach` als Argument hinter dem Prädikat `ic(Von,Nach)` im Regelrumpf aufführen.

```
anfrage:-write(' Gib Abfahrtsort: '),nl,
        ttyread(Von),
        write(' mögliche(r) Ankunftsort(e): '),nl,
        ic(Von,Nach),
        write(Nach),nl.
anfrage:-nl,write(' Ende ').
```

```
Gib Abfahrtsort:
kö.
mögliche(r) Ankunftsort(e):
ka
yes
```

- Eine Antwort wird geliefert.
- Wie erzwingen wir alle Antworten ohne einen neuen Programmstart?
- Wir lösen dieses Problem durch den Einsatz des argumentlosen Standard-Prädikats `fail`, bei dem jeder Unifizierungs Versuch fehlschlägt.
- `:- anfrage,fail.`

- Dadurch wird - sofern das 1. Subgoal `anfrage` erstmals abgeleitet wurde - durch das 2. Subgoal `fail` ein Backtracking erzwungen, d.h. es wird zur letzten Backtracking-Klausel - also zur abgeleiteten Klausel mit dem Klauselkopf `ic(Von,Nach)` im Regelrumpf der 1. Regel des Prädikats `anfrage` - zurückgekehrt und von dort aus versucht, das Prädikat `ic(kö,Nach)` mit einer alternativen Instanzierung der Variablen `Nach` erneut zu unifizieren.

- Ist dieser Unifizierungs-Versuch wiederum erfolgreich, so wird der instanzierte Wert der Variablen `Nach` als nächster, möglicher Ankunftsort angezeigt und daraufhin durch das Prädikat `fail` ein erneutes Backtracking erzwungen.
- Dieser Prozess wird solange wiederholt, bis es keine weiteren Instanzierungen der Variable `Nach` mehr gibt, mit der sich das Prädikat `ic(kö,Nach)` ableiten läßt.
- Daraufhin wird ein Backtracking beim Prädikat `anfrage` durchgeführt, so dass der Text `Ende` angezeigt und die Ableitbarkeits - Prüfung des internen Goals `anfrage,fail.` erfolglos beendet wird.

```
dic(ha,kö).
dic(ha,fu).
dic(kö,ka).
dic(kö,ma).
dic(fu,mü).
dic(ma,fr).

ic(Von,Nach):-dic(Von,Nach).
ic(Von,Nach):-dic(Von,Z),ic(Z,Nach).

anfrage:-write(' Gib Abfahrtsort: '),nl,
        ttyread(Von),
        write( ' mögliche(r) Ankunftsort(e): '),nl,
        ic(Von,Nach),
        write(Nach),nl.
anfrage:-nl,write( ' Ende ').

:- anfrage,fail.
```

Erweiterung 4

- Es sollen alle möglichen IC-Verbindungen zwischen allen möglichen Abfahrts- und Ankunftsorten angezeigt werden.
- Die erste Regel wird geändert:
-
```
anfrage:-write( mögliche IC-Verbindung(en):
),nl,
ic(Von,Nach),
write( Abfahrtsort: ),write(Von),nl,
write( Ankunftsort: ),write(Nach),nl.
```

- Läßt sich nämlich eine IC-Verbindung aus der Wiba ableiten, so wird diese IC-Verbindung unmittelbar in die dynamische Wiba eingetragen.
- Dabei wird nicht geprüft, ob der abgeleitete Fakt bereits Bestandteil der dynamischen Wiba ist.

Aufgabe

Eine IC-Verbindung, die bereits als Fakt in der dynamischen Wiba enthalten ist, soll nicht erneut in die dynamische Wiba eingetragen werden. Außerdem sollen bereits abgeleitete IC-Verbindungen der dynamischen Wiba entnommen und nicht von neuem aus der statischen Wiba abgeleitet werden.

Dynamische Wiba sichern

- Ein abgeleitetes Faktum aus der statischen Wiba kann in einer dynamischen Wiba aufgenommen werden, um es für weitere Inferezen bereitzustellen.
- Für die Übernahme von Fakten in die dynamische Wiba stehen die Standard-Prädikate `asserta` und `assertz` zur Verfügung.
- Bei der Unifizierung wird das Argument von `asserta` (`assertz`) vor (hinter) allen anderen Fakten in die dynamische Wiba eingefügt.
- Soll eine Regel in die dynamische Wiba eingefügt werden, so sind diese Prädikate mit 2 Argumenten - zur Angabe des Regelkopfs und des Regelrumpfs - zu verwenden.

Dynamische Wiba sichern

- Für das Eintragen bereits abgeleiteter IC-Verbindungen in die dynamische Wiba und den späteren Zugriff auf diese IC-Verbindungen führen wir ein neues Prädikat ein `ic_db`.
- Durch den Einsatz des Standard-Prädikats `asserta` läßt sich die Tatsache, dass `ic(ha,fr)` aus der (statischen) Wiba ableitbar ist, in der Form
  `asserta(ic db(ha,fr))`
  als Fakt in die dynamische Wiba eintragen.
- Bei der Unifizierung des Standard-Prädikats `asserta` wird der Fakt ic_db(ha,fr). in die dynamische Wiba übernommen.
- Bei einer späteren Anfrage nach einer IC-Verbindung von `ha` nach `fr` kann diese Anfrage mit dem in die dynamische Wiba eingetragenen Fakt `ic_db(ha,fr).` in Verbindung gebracht werden, so dass keine erneute, schrittweise Ableitung vorgenommen werden muss.

- Wir erweitern nun unsere Prolog-Programme, so dass abgeleitete Fakten aus der Wiba mitgenutzt werden.

```
anfrage:-write( Gib Abfahrtsort: ),nl,
read(Von),
write( Gib Ankunftsort: ),nl,
read(Nach),
verb(Von,Nach),
write( IC-Verbindung existiert ),nl,
asserta(ic db(Von,Nach)).
anfrage:-write( IC-Verbindung existiert nicht
),nl.
```

- Falls eine IC-Verbindung über eine oder mehrere Zwischenstationen ableitbar ist, wird anschließend das Standard-Prädikat `asserta` unifiziert und folglich das im Argument aufgeführte Prädikat als Fakt mit dem Prädikatsnamen `ic_db` und den Zwischenstationen als instanzierten Variablenwerten in die dynamische Wiba eingetragen.

PRÄDIKAT VERB

- Suchen in der statischen *oder* dynamischen Wiba.
- Durch die 1. Regel wird eine Ableitung aus der dynamischen Wiba und durch die 2. Regel eine Ableitung aus der statischen Wiba versucht.

```
verb(Von,Nach):-ic db(Von,Nach),
write( ableitbar aus dynamischer Wiba ),nl.
verb(Von,Nach):-ic(Von,Nach).
```

- Wollen wir uns anschließend die jeweils abgeleiteten IC-Verbindungen, die als Fakten in die dynamische Wiba aufgenommen wurden, anzeigen lassen, so setzen wir das Standard-Prädikat `listing` ein:
  ```
  ? listing(ic_db).
  ```
- Durch die Unifizierung dieses Prädikats werden alle Klauseln mit dem Prädikatsnamen `ic_db` angezeigt, die aktuell in der Wiba enthalten sind.

- Problem: Jedes Mal wenn wir das Programm beenden, geht der Inhalt der Wiba verloren.
- Aufgabe: Sicherung der dynamischen Wiba in einer Sicherungs-Datei.
- Lösung: Einsetzen der Prädikate `tell, listing, told.`

```
sichern_ic:-ic db(_ ,_ ), tell(ic.pl),
listing(ic db), told.
sichern_ic.
```

- Wir überprüfen vor einer Übertragung, ob in der Wiba überhaupt Klauseln mit dem Prädikatsnamen `ic_db` vorhanden sind.
- Durch die Unifizierung des Prädikats `tell(ic.pl)` werden alle nachfolgenden Ausgaben des PROLOG-Systems solange in die Datei `ic.pl` ubertragen, bis das Prädikat `told` unifiziert wird.
- Da durch die vorausgehende Unifizierung des Prädikats `listing(ic_db)` alle Klauseln mit dem Prädikatsnamen `ic_db` angezeigt werden, wird durch die Ableitung des Prädikats `sichern_ic` die Übertragung aller Klauseln mit dem Prädikatsnamen `ic_db` in die Sicherungsdatei `ic.pl` durchgeführt.

- Um den Inhalt einer Sicherungs-Datei in die aktuelle Wiba zu übernehmen, setzen wir das Standard-Prädikat `consult` in der Form
  `consult(ic.pl)`
  ein.
- Durch die Unifizierung dieses Prädikats werden zunächst alle diejenigen Klauseln aus der aktuellen Wiba gelöscht, für die gleichnamige Klauselköpfe in der Sicherungs-Datei `ic.pl` enthalten sind.
- Anschließend wird die Wiba um den Inhalt von `ic.pl` ergänzt

- Vor dem Zugriff auf den Inhalt einer Sicherungs-Datei sollte überprüft werden, ob die Datei auch tatsächlich vorhanden ist.
- Dazu läßt sich das Standard-Prädikat `exist_file` in der Form
  `exist_file(ic.pl).`
  einsetzen.
- Dieses Prädikat läßt sich in dieser Form nur dann unifizieren, wenn die Datei `ic.pl` existiert und auf ihren Inhalt ein lesender Zugriff erlaubt ist.

- Fur die Ubertragung der Fakten aus der Sicherungs-Datei `ic.pl` vereinbaren wir die folgenden Klauseln:

```
lesen:-exists('ic.pl),
consult(ic.pl).
lesen.
```

- Die zweite Klausel haben wir deswegen angegeben, damit in der Situation, in der die Datei `ic.pl` noch nicht existiert oder auf ihren Inhalt kein Lese-Zugriff erlaubt ist, der Regelkopf trotzdem - durch Backtracking - ableitbar ist.
- Das interne Goal wird zu

```
:- lesen,anfrage,sichern_ic.
```

- Der Nachteil dieses Programms besteht darin, dass jedesmal, wenn eine Anfrage nach einer IC-Verbindung positiv beantwortet wird, ein Eintrag in die dynamische Wiba durch das Standard-Prädikat `asserta` vorgenommen wird.
- Dies führt dazu, dass die dynamische Wiba bei jeder erfolgreichen Ableitung um einen Fakt erweitert wird, so dass bereits abgeleitete IC-Verbindungen mehrfach in die dynamische Wiba eingetragen werden.
- Dies kann man mit Hilfe des Prädikats `cut` unterbinden.

Anfrage

- Eine IC-Verbindung, die bereits als Fakt in der dynamischen Wiba enthalten ist, wird nicht erneut in die dynamische Wiba eingetragen.
- Ausserdem sollen bereits abgeleitete IC-Verbindungen der dynamischen Wiba entnommen und nicht von neuem aus der statischen Wiba abgeleitet werden .
- Dazu muss das Prädikat `anfrage` geeignet geändert werden.
- Wir verwenden dazu das Prädikat `not`: `\+`.

- `Not` ist genau dann ableitbar, wenn das als Argument aufgeführte Prädikat in der Wiba nicht unifiziert werden kann.
- Das Prädikat `not` kann nicht dazu eingesetzt werden, um für ein Prädikat Variablen-Instanzierungen zu bestimmen, die sich nicht aus der Wiba ableiten lassen.

- Ist also das Prädikat `verb(Von,Nach)` ableitbar, so ist - vor einem Eintrag in die dynamische Wiba - zu prüfen, ob das Prädikat `ic_db(Von,Nach)` für die aktuellen Instanzierungen der Variablen `Von` und `Nach` bereits als Fakt in der dynamischen Wiba enthalten ist.
- Dazu fügen wir das Prädikat `\+(ic_db(Von,Nach))` wie folgt in die oben angegebenen Regeln ein:

```
anfrage:-write( Gib Abfahrtsort: ),nl,
read(Von),
write( Gib Ankunftsort: ),nl,
read(Nach),
verb(Von,Nach),
write( IC-Verbindung existiert ),nl,
\+(ic_db(Von,Nach)),
asserta(ic db(Von,Nach)).
anfrage:-write( IC-Verbindung existiert nicht
),nl.
```

- In dieser Situation läßt sich das Prädikat \+ nur dann unifizieren, wenn `ic_db(Von,Nach)` mit den aktuellen Instanzierungen der Variablen `Von` und `Nach` nicht aus der dynamischen Wiba ableitbar ist.
- In diesem Fall wird das Prädikat `ic_db(Von,Nach)` (mit den aktuellen Instanzierungen der Variablen `Von` und `Nach`) durch die nachfolgende Unifizierung des Standard-Prädikats `asserta` als Fakt in die dynamische Wiba aufgenommen.

- Ist dagegen `ic_db(Von,Nach)` - z.B. für die Instanzierungen der Variablen `Von` mit `ha` und der Variablen `Nach` mit `mu` - bereits als Fakt in der dynamischen Wiba enthalten, so kann `ic_db(ha,mu)` unifiziert und damit `\+(ic_db(ha,mu))` nicht abgeleitet werden.
- Daraufhin setzt Backtracking ein.
- Dies führt dazu, dass das Prädikat `verb(ha,mu)` - durch ein daraufhin eingeleitetes Backtracking - mit dem Kopf der 2. Regel des Prädikats `verb(Von,Nach)`
  `verb(Von,Nach):-ic(Von,Nach).`
  unifiziert wird, woraufhin die Ableitbarkeit des Prädikats `ic(ha,mu)` aus den Fakten der (statischen) Wiba erneut nachgewiesen wird.

- Im Anschluss daran erweist sich wiederum das Subgoal \+(ic_db(ha,mu)) als nicht unifizierbar.
- Da kein weiteres Backtracking innerhalb der 1. Regel (mit dem Prädikat `anfrage` im Regelkopf) mehr möglich ist, erfolgt Backtracking zur 2. Regel mit dem Prädikat `anfrage` im Regelkopf, woraufhin der Text `IC-Verbindung existiert nicht` angezeigt wird.
- Dies ist jedoch nicht das von uns erwunschte Ergebnis.

- Damit die Ableitbarkeits-Prüfung nicht in der eben beschriebenen Form durchgeführt wird, muss das Backtracking zum Subgoal `verb` und das anschließende Backtracking zur 2. Regel des Prädikats `anfrage`, das durch die Nicht-Ableitbarkeit des Prädikats `\+` erzwungen wird, verhindert werden.
- Zur Einschränkung des Backtrackings steht das Standard-Prädikat cut zur Verfügung, das in einem PROLOG-Programm durch das Ausrufungszeichen ! gekennzeichnet wird.

- Das Prädikat cut läßt sich beim ersten Ableitbarkeits-Versuch - von links kommend - erfolgreich unifizieren.
- Wird das Prädikat cut unifiziert, so sind alle alternativen Regeln - mit dem gleichen Pr adikat im Regelkopf - durch Backtracking nicht mehr erreichbar (sie sind gesperrt).
- Dies bedeutet, dass keine Alternativen zur aktuellen Regel betrachtet werden können, so dass das aktuelle Parent-Goal nur über die aktuelle Regel und nicht über alternative Regeln ableitbar ist.

- Folgt dem cut ein weiteres Prädikat innerhalb einer logischen UND Verbindung, und schlägt die Ableitbarkeits-Prüfung dieses Prädikats fehl, so ist, wenn durch Backtracking das Prädikat cut (*von rechts kommend*) erneut erreicht wird, das Pr adikat cut bei diesem 2. Ableitbarkeits-Versuch nicht unifizierbar.
- Demzufolge ist kein Backtracking zu einem*links* vom cut stehenden Prädikat mehr möglich, so dass das Parent-Goal nicht ableitbar ist.

- Das Prädikat cut wirkt also wie eine Mauer, die *von links kommend* überwunden werden kann.
- Wird diese Mauer anschließend *von rechts kommend* - durch Backtracking - erreicht, so kann sie nicht *übersprungen* werden.
- In dieser Situation werden keine Alternativen für ein Backtracking mehr berücksichtigt, so dass das Parent-Goal nicht unifizierbar ist.

```
anfrage:-write( Gib Abfahrtsort: ),nl,
read(Von),
write( Gib Ankunftsort: ),nl,
read(Nach),
verb(Von,Nach),
write( IC-Verbindung existiert ),nl,
!,
\+(ic db(Von,Nach)),
asserta(ic db(Von,Nach)).
anfrage:-write( IC-Verbindung existiert nicht
),nl.
```

- Ist nämlich das Subgoal `\+(ic_db(Von,Nach))` nicht unifizierbar, so verhindert der cut ein Backtracking zum Subgoal `verb(Von,Nach)` und damit auch ein Backtracking zur 2. Klausel des Prädikats `anfrage`.
- Damit ist der Rumpf der 1. Regel nicht ableitbar.
- Da die 2. Regel durch das erstmalige Ableiten des cut gesperrt ist, kann demzufolge der Regelkopf mit dem Prädikat `anfrage` (als Parent-Goal) nicht abgeleitet werden.

- Internes Goal: `:- lesen,!,anfrage,sichern_ic.`
- Dies ist erforderlich, weil sichergestellt werden muss, dass kein Backtracking beim Scheitern der Ableitbarkeits-Prüfung des Subgoals `anfrage` zum Subgoal `lesen` durchgefuhrt wird.
- Dies hätte zur Folge, dass ein Backtracking beim Prädikat `lesen` durchgefuhrt würde, sofern die Datei `ic.pl` beim Start des Programms vorhanden ist.
- Weil in diesem Fall das Prädikat `lesen` erfolgreich abgeleitet werden kann, wurde für das Prädikat `anfrage` eine erneute - nicht erwünschte - Ableitbarkeits-Prüfung durchgeführt.

- Das Standard-Prädikat cut hat, nachdem es einmal abgeleitet wurde, Auswirkungen auf die weitere Arbeitsweise des Inferenz-Algorithmus.
- Es besitzt einen Seiteneffekt, da es (nachfolgende) alternative Klauseln für Backtracking sperrt, so dass das Parent-Goal nicht über alternative Klauseln ableitbar ist.

Beispiel

```
b.
c:-fail.
d.
a:-!,b.
a:-d.
test:-a,c.
```

- Bei der Ableitbarkeits-Prüfung des Goals `test.` wird zunächst das 1. Subgoal `a` mit der 1. Regel mit dem Regelkopf `a` unifiziert.
- Anschließend wird das Standard-Prädikat `cut` unifiziert und dadurch die 2. Klausel mit dem Prädikat `a` im Klauselkopf für Backtracking gesperrt.
- Da das Prädikat `b` als Fakt auftritt, ist das 1. Subgoal `a` ableitbar.
- Da das 2. Subgoal `c` im Regelrumpf des Prädikats `test` - wegen der Regel `c:-fail.` - nicht ableitbar ist, musste Backtracking zur 2. Regel mit dem Prädikat `a` im Regelkopf durchgeführt werden.
- Dies ist jedoch nicht möglich, weil dieses Backtracking zuvor durch den `cut` gesperrt wurde, so dass folglich das externe Goal `test.` nicht abgeleitet werden kann.

Unterbinden des Backtrackings mit den Prädikaten cut und fail

Cut-fail-Kombination

- Fail → erschöpfendes Backtracking
- Cut → eingeschränktes Backtracking
- Oftmals besteht Interesse, im Rahmen der Ableitbarkeits-Prüfung das Scheitern eines Parent-Goals herbeizuführen, indem ein noch mögliches Backtracking unterbunden wird.
- Dazu müssen wir eine Kombination der Prädikate cut und fail als cut-fail in der Form von
  !,fail.
  einsetzen.

Unterbinden des Backtrackings mit den Prädikaten cut und fail

Cut-fail-Kombination

- Wird diese Prädikat-Kombination bei der Ableitbarkeits-Prüfung einer UND-Verbindung erreicht, so wird der Regelrumpf und damit auch der zugehörige Regelkopf, d.h. das Parent-Goal, als nicht ableitbar erkannt.

Anwendung

- Es ist ein Programm zu entwickeln, das solange eine Eingabe von der Tastatur anfordert, bis die Eingabe einer bestimmten Text-Konstanten erfolgt.

Beispiel

```
auswahl(s):-nl,write( Ende ).
auswahl(X):-write( Eingabe von: ),write(X),nl,fail.
anforderung:-write( Gib Buchstabe (Abbruch bei Eingabe von s
):),nl,
read(Buchstabe),
auswahl(Buchstabe),
!,fail.
anforderung:-anforderung.
:- anforderung.
```

Programmlauf

- Programm startet, wir geben e ein.
- Regelkopf auswahl(X) (durch die Instanzierung X:=e) wird unifiziert.
- Die Ableitbarkeits-Prufung des Regelrumpfs schlägt fehl, da er durch das Prädikat fail abgeschlossen wird.

Programmlauf

- Da beim Ableiten des internen Goals im 1. Regelrumpf des Prädikats anforderung das Prädikat cut in diesem Regelrumpf noch nicht erreicht wurde, erfolgt Backtracking zur 2. Regel mit dem Regelkopf anforderung.
- Anschlieëend wird erneut - mit einem neuen Exemplar der Wiba - der Rumpf der 1. Regel mit dem Kopf anforderung auf Ableitbarkeit untersucht.
- Dies führt durch die Unifizierung des Subgoals auswahl(Buchstabe) zu einer neuen Eingabeanforderung.

Programmlauf

- Erst wenn der Buchstabe s erstmalig eingegeben wird, erfolgt die Unifizierung des Prädikats auswahl(s), so dass die Prädikat-Kombination cut-fail erreicht wird.
- Da der Regelrumpf - wegen des Prädikats fail - nicht ableitbar ist und Backtracking zur 2. Regel des Prädikats anforderung durch das Prädikat cut verhindert wird, ist das Parent-Goal anforderung und somit das interne Goal nicht ableitbar.
- Folglich läßt sich die Programmschleife durch die Eingabe des Buchstabens s abbrechen.

Rote und grüne Cuts

1. kopf :- a , ! , b .
2. kopf :- c , ! .
3. kopf :- d , ! , fail .
4. kopf :- ! , e .

Ableitung von “a” möglich:				
	Ableitung von “b” möglich	Parent-Goal ableitbar ((2),(3) und (4) sind für (seichtes) Backtracking gesperrt)		
	Ableitung von “b” *nicht* möglich	Parent-Goal *nicht* ableitbar ((2),(3) und (4) sind für (seichtes) Backtracking gesperrt)		

<table>
<tr><td colspan="4">Ableitung von “a” nicht möglich:</td></tr>
<tr><td></td><td>Ableitung von “c” möglich</td><td>Parent-Goal ableitbar ((3) und (4) sind für (seichtes) Backtracking gesperrt)</td><td></td></tr>
<tr><td></td><td>Ableitung von “c” nicht möglich</td><td>(seichtes) Backtracking zu (3)</td><td></td></tr>
<tr><td colspan="4">Ableitung von “c” nicht möglich:</td></tr>
<tr><td></td><td></td><td>Ableitung von “d” möglich</td><td>Parent-Goal nicht ableitbar ((4) ist für (seichtes) Backtracking gesperrt)</td></tr>
<tr><td></td><td></td><td>Ableitung von “d” nicht möglich</td><td>(seichtes) Backtracking zu (4)</td></tr>
<tr><td colspan="4">Ableitung von “d” nicht möglich:</td></tr>
<tr><td></td><td></td><td></td><td>Ableitung von “e” möglich</td><td>Parent-Goal ableitbar</td></tr>
<tr><td></td><td></td><td></td><td>Ableitung von “e” nicht möglich</td><td>Parent-Goal nicht ableitbar</td></tr>
</table>


Wann wird cut eingesetzt?

- Es soll sichergestellt werden, dass nur dann die Möglichkeit bestehen darf, das Parent-Goal durch die aktuell betrachtete Regel abzuleiten, wenn innerhalb des zugehörigen Regelrumpfs ein bestimmtes Prädikat ableitbar ist.
- Dazu muss das Prädikat cut unmittelbar links von diesem Prädikat eingefugt werden, so dass nachfolgende alternative Regeln - durch Backtracking - nicht mehr untersucht werden können.
- Das bedeutet insbesondere, dass die vor dem Ableiten des Prädikats cut eingegangenen Instanzierungen von Variablen für die weiteren Ableitbarkeits-Prüfungen, die hinter dem Prädikat cut durchgeführt werden, nicht wieder gelöst werden können (sie sind eingefroren).

Roter Cut

- Ein derartiger Einsatz des Prädikats cut ändert die deklarative und prozedurale Bedeutung des Prolog-Programms.
- Roter Cut

Prozedurale Bedeutung eines Prolog-Programms

- Wird bestimmt durch den Ablauf der Ableitbarkeitsprüfung.
- Der Inferenz-Algorithmus kann bei einer veränderten Reihenfolge der Klauseln oft auch einen anderen Ablauf nehmen, kann sich die prozedurale Bedeutung eines PROLOG-Programms entsprechend ändern $\rightarrow$ endlos Schleife.

Deklarative Bedeutung eines Prolog-Programms

- Wird durch die Prädikate in den einzelnen Klauseln bestimmt, unabhängig von der Reihenfolge in der die Klauseln angegeben sind.
- Die deklarative Bedeutung besteht somit darin, ob und nicht wie ein Goal ableitbar ist.

Zusammenfassung: Deklarative und prozedurale Bedeutung

Der deklarative Aspekt eines Prolog Programms betrifft nur die Relationen, die im Programm definiert sind.
Der prozedurale Aspekt betrifft die Art und Weise wie der Output des Programms zustande kommt.

Vorteil von Prolog:

übernimmt weite Teile der prozeduralen Aspekte selbst. Damit kann sich der Programmierer auf die normalerweise leichter zu verstehenden deklarativen Aspekte konzentrieren. Späterhin kommt man jedoch nicht umhin sich auch mit dem internen Programmablauf zu befassen und diesen etwa durch geschickte Anordnung der Klauseln oder direkte Eingriffe wie den Cut zu beeinflussen.

DEKLARATIV - WAS

Gegeben ein Programm und ein goal G, dann ist G wahr (oder aus dem Programm erfüllbar) gdw.

1. es existiert eine Klausel C im Programm, so dass
2. es existiert eine Instanz I von C, so dass
   1. das Kopf von I ist identisch mit G, und
   2. alle goals aus dem Rumpf von I wahr sind.

PROZEDURAL - WIE

Die Beantwortung einer Frage = Liste von goals zu erfüllen: die Variablen werden so instantiert, dass alle goals logisch aus dem Programm folgen.
Die prozedurale Steuerung in Prolog ist eine Prozedur für eine Liste von goals.
EXECUTE bedeutet: versuche die goals zu erfüllen.

Grüner Cut

- Neben der Unterbindung des Backtrackings, weil nur so die gegebene Aufgabenstellung gelöst werden kann, gibt es einen weiteren Grund, das Prädikat cut in einem PROLOG-Programm oder einem Goal einzusetzen.
- Oftmals kann durch das Prädikat cut eine Effizienzsteigerung bei der Ausführung eines Programms erreicht werden, indem verhindert wird, dass alternative Lösungen weiter untersucht werden.
- Grüner Cut.

Grüner Cut

- Bei einem grünen Cut wird die prozedurale Bedeutung eines PROLOG-Programms, nicht jedoch die deklarative Bedeutung geändert.

```
anfrage:-write( Gib Abfahrtsort: ),nl,
read(Von),
dic(Von,_ ),
!,
write( Es gibt eine Direktverbindung: ),nl.
anfrage:-write( Es gibt keine Direktverbindung
),nl,
?-anfrage.
```

- Damit kann man die Behauptung
  es gibt mindestens eine Direktverbindung von `ha` aus
  beantworten.

- Durch den Einsatz des Prädikats cut - nachdem die erste Instanzierung der Variablen `X` mit der Konstanten `kö` vorgenommen wurde - werden keine weiteren Ableitbarkeits-Prüfungen mehr durchgeführt.
- Setzen wir das Prädikat cut in dieser Form ein, so gehen keine Lösungen verloren, da wir lediglich an der Existenz mindestens einer Direktverbindung von `ha` aus interessiert sind.

- Grundsätzlich ist der Einsatz des Prädikats cut nicht unproblematisch.
- Es ist stets die jeweils zugrundeliegende Anfrage zu beachten, zu deren Lösung das jeweilige PROLOG-Programm entwickelt wurde.
- Der Einsatz des Prädikats cut - in der Wirkung eines roten Cut - kann leicht dazu führen, dass mögliche Lösungen durch unterbundenes seichtes und tiefes Backtracking nicht abgeleitet oder aber falsche Ergebnisse angezeigt werden.

Beispiel 1

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

?-max(5,10,Maximum).
```

Beispiel 1

- `Maximum = 10;`
  `no`
- Dabei wird die Anfrage mit der 1. Klausel erfolgreich abgeleitet, so dass die Variable Maximum bzw. Y mit dem größeren der beiden Werte instanziert ist.
- Da die Bedingung `X =< Y` erfüllt ist, wird anschließend das Prädikat cut abgeleitet und somit Backtracking zur 2. Klausel unterbunden.

Beispiel 1

- Entfernen wir den Cut in der 1. Klausel und stellen die gleiche Anfrage, so erhalten wir das gleiche Ergebnis angezeigt.
- Dies liegt daran, dass jetzt - nach der Eingabe eines Semikolons - Backtracking zur 2. Klausel stattfindet.
- Die Ableitung der Anfrage mit der 2. Klausel scheitert jedoch, da die Bedingung `10 =< 5` nicht erfüllt ist.
- Da wir in beiden Programmversionen das gleiche (erwartete) Ergebnis erhalten, liegt ein grüner Cut vor.

Beispiel 2

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

?- -max(5,10,Maximum).
```

Beispiel 2

- Entfernen wir den Cut in der 1. Klausel und stellen wiederum die gleiche Anfrage, so wird die Anfrage mit einem falschen Ergebnis in der Form

```
Maximum = 10;
Maximum = 5;
no
```

beantwortet.

Beispiel 2

- Bei der Ableitbarkeits-Prufüng wird die Anfrage zunächst mit der 1. Klausel und - nach der Eingabe des Semikolons - auch mit der 2. Klausel erfolgreich abgeleitet.
- Somit erhalten wir für die Variable `Maximum` die beiden Werte `10` und `5` angezeigt.
- Da keine weitere Instanzierung der Variablen `Maximum` mehr möglich ist, führt die Eingabe eines weiteren Semikolons zur abschließenden Ausgabe des Textes `no`.
- Da wir in beiden Programmversionen nicht das gleiche Ergebnis, sondern in der Version ohne Einsatz des Cut ein unerwartetes und falsches Ergebnis angezeigt bekommen, handelt es sich hier um einen roten Cut.

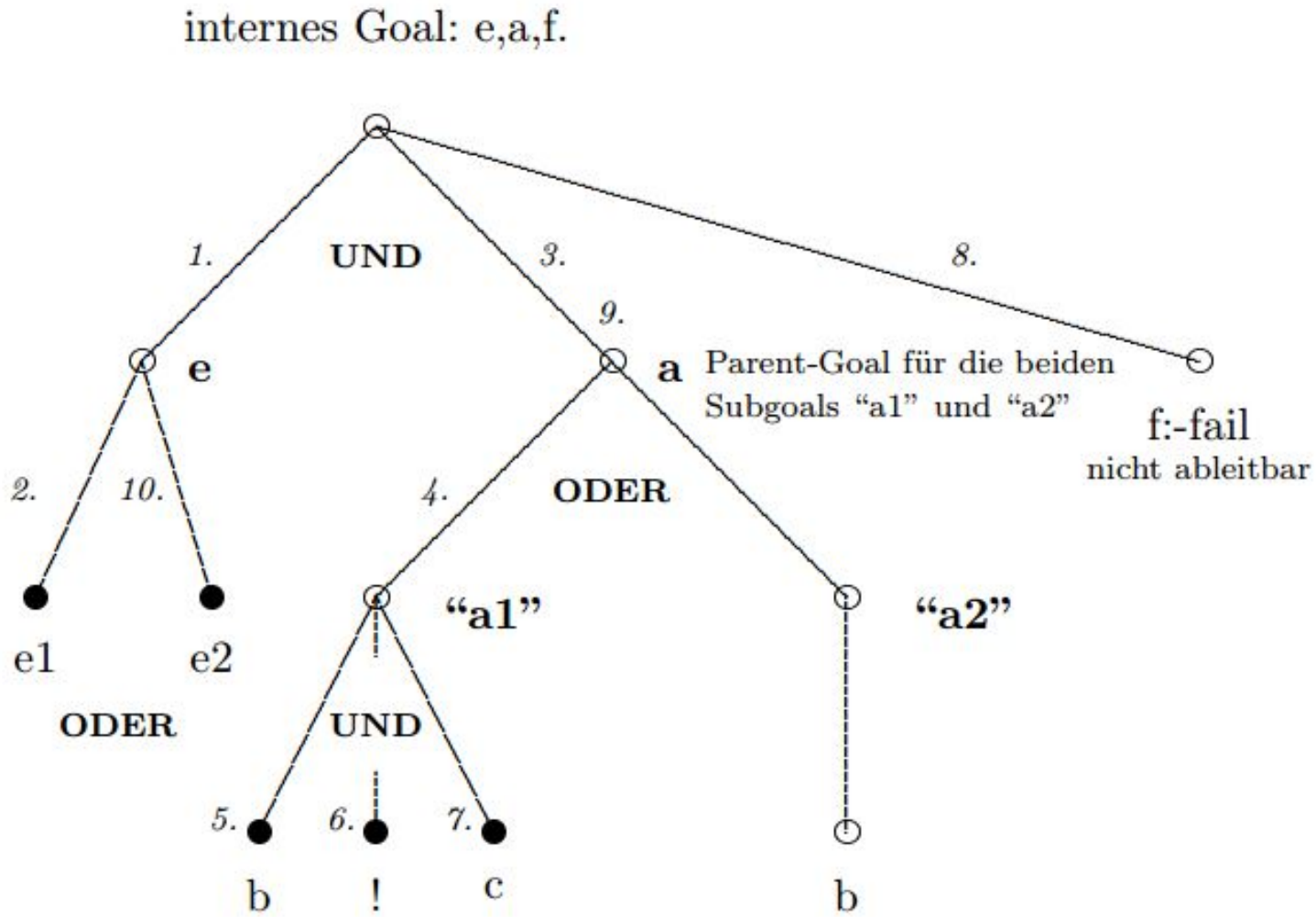
Cut

- Das Standard-Prädikat cut ist das einzige Prädikat, das die Arbeit der Inferenzkomponente direkt beeinflusst.
- Die Farbe des Cut läßt sich stets dadurch bestimmen, dass eine Anfrage an das ursprüngliche PROLOG-Programm und anschließend die gleiche Anfrage an das gleiche Programm - ohne den Cut - gestellt wird.
- Führt die Anfrage an das derart geänderte Programm zu unerwarteten oder gar falschen Ergebnissen, so liegt ein roter Cut vor.
- Ansonsten handelt es sich um einen grünen Cut.

Beispiel 3: Demonstration des Prädikats cut

```
b.
c.
e1.
e2.
e:-e1.
e:-e2.
f:-fail.
a:-b,!,c.
a:-b.
:-e,a,f.
```

Beispiel 3: Demonstration des Prädikats cut



Beispiel 3: Demonstration des Prädikats cut

- Zur Überprüfung der Ableitbarkeit des internen Goals wird als erstes das Subgoal `e` abgeleitet, weil das Prädikat `e1` unifizierbar ist.
- Anschließend muss die Unifizierbarkeit des 2. Subgoals `a` - als Parent-Goal für die beiden zugehörigen Subgoals `b,!,c` (im Ableitungsbaum ist dieses Parent-Goal durch `a1` gekennzeichnet) und `b` (im Ableitungsbaum durch `a2` gekennzeichnet) in den Regelrumpfen der beiden Regeln mit dem Regelkopf `a` - untersucht werden.

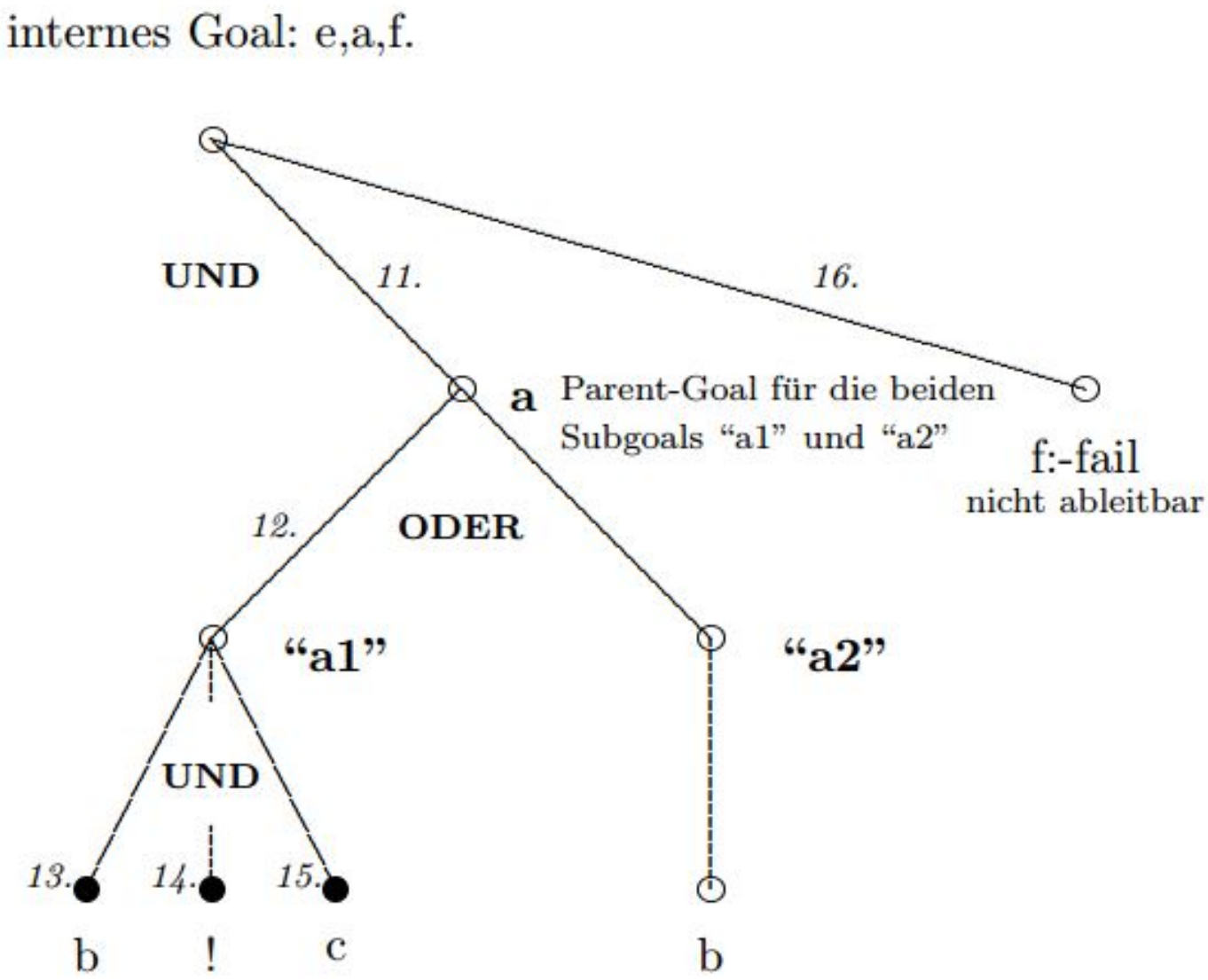
Beispiel 3: Demonstration des Prädikats cut

- Dazu wird in der 1. Regel - mit dem Prädikat `a` im Regelkopf zunächst das Prädikat `b` und anschließend das Prädikat cut unifiziert, so dass die 2. Regel `a:-b.` für jeden zu einem späteren Zeitpunkt einsetzenden Unifizierungs-Versuch des Parent-Goals `a` gesperrt ist.
- Da das Prädikat `c` in der Regel `a:-b,!,c.` ebenfalls ableitbar ist, erweist sich der gesamte Regelrumpf und damit der Regelkopf `a` als ableitbar.
- Anschließend wird das 3. Subgoal `f` des internen Goals überprüft.
- Wegen der Nicht-Ableitbarkeit des Subgoals `f` erfolgt Backtracking.

Beispiel 3: Demonstration des Prädikats cut

- Da ein Backtracking zur 2. Regel `a:-b.` - wegen der vorausgegangenen Unifizierung des Standard-Prädikats `cut` - nicht mehr möglich ist, wird das Parent-Goal `a` als nicht ableitbar erkannt.
- Dadurch wird ein Backtracking im internen Goal zum Prädikat `e` vorgenommen, für das die 1. Regel `e:-e1.` als Backtracking-Klausel markiert ist.
- Nach Backtracking erweist sich der Regelrumpf der 2. Regel `e:-e2.` als ableitbar.
- Somit wird ein erneuter Unifizierungs-Versuch des 2. Subgoals `a` im internen Goal versucht.
- Da hierbei mit einem neuen Exemplar der Wiba gearbeitet wird, hat folglich die ursprünglich eingetretene Wirkung des Standard-Prädikats cut, das nur noch Backtracking hinter dem cut zuließ, keine Bedeutung.

Beispiel 3: Demonstration des Prädikats cut



Beispiel 3: Demonstration des Prädikats cut

- Diese erneute Ableitbarkeits-Prüfung für das 2. Subgoal mit dem Prädkat `a` wird genauso vorgenommen, wie wir es oben für den 1. Versuch beschrieben haben.
- Folglich ist das Subgoal `a` (im Schritt 15) wiederum unifizierbar.
- Der nachfolgende Versuch der Unifizierung des 3. Subgoals `f` schlägt wieder fehl, so dass Backtracking versucht wird.
- Da wiederum - wegen des bereits unifizierten Prädikats cut - kein Backtracking für das 2. Subgoal `a` durchgeführt werden kann und zusätzlich auch keine Alternative für ein Backtracking zum Ableiten des 1. Subgoals `e` mehr existiert, wird (im 16. Schritt) endgültig das Scheitern des internen Goals festgestellt.