Zusammenfassung Negation

- keine echte logische Negation: stattdessen Negation as Failure
- Beweissuche in „Nebenrechnung“, bindet nach außen keine Variablen
- lässt sich nur prozedural/operationell (wirklich) verstehen
- Probleme bei Versuch deklarativer Lesart:
  - nicht kompositionell/substitutiv
  - sensitiv gegenüber Änderungen der Reihenfolge innerhalb Regeln
  - T_P-Operator wäre nicht-monoton

Deskriptive Programmierung

Der Cut-Operator

Der „Cut“-Operator (1)

- Die operationelle **Backtracking-Methode** von Prolog:
  - merkt sich jeden Punkt, an dem noch weitere Alternativen (durch andere Regeln) besucht werden könnten.
  - kann dadurch einen hohen Verwaltungs- und Ausführungsaufwand erfordern.

- Durch Verwendung des **„Cut“-Operators**: **!**
  - kann das Backtracking explizit beeinflusst werden,
  - nämlich das Ausprobieren bestimmter **Alternativen verhindert** werden, indem Teile des Ableitungsbaums „abgeschnitten“ werden.
  - können daher die Laufzeit und der Speicherbedarf verringert werden.
  - können Programme unter Umständen „einfacher“/kürzer werden.
  - können **korrekte** Antworten unter Umständen nicht mehr gefunden werden.

Der „Cut“-Operator (2)

- Der Cut wird in Anfragen oder Regelrümpfen wie ein Literal mit dem 0-stelligen Prädikat **!** notiert.

- Die Bedeutung des „Cut“-Operators kann nur **operationell** angegeben werden.

- Beispiel:

```
p(X) :- q(X),!,s(X).
p(X) :- r(X).
```

  - Kann `q(X)` ***nicht*** bewiesen werden, wird für `p(X)` die nächste Klausel ausprobiert.

  - Wird `q(X)` ***einmal*** bewiesen, wird wegen **!** im Fall eines späteren Fehlschlags, in `s(X)`, kein weiterer Beweis für `q(X)` gesucht. Es wird dann auch keine weitere Regel für `p(X)` mehr ausprobiert.

    Also, ist `s(X)` mit der durch `q(X)` ermittelten Belegung für `X` nicht beweisbar, dann ist auch `p(X)` nicht beweisbar.

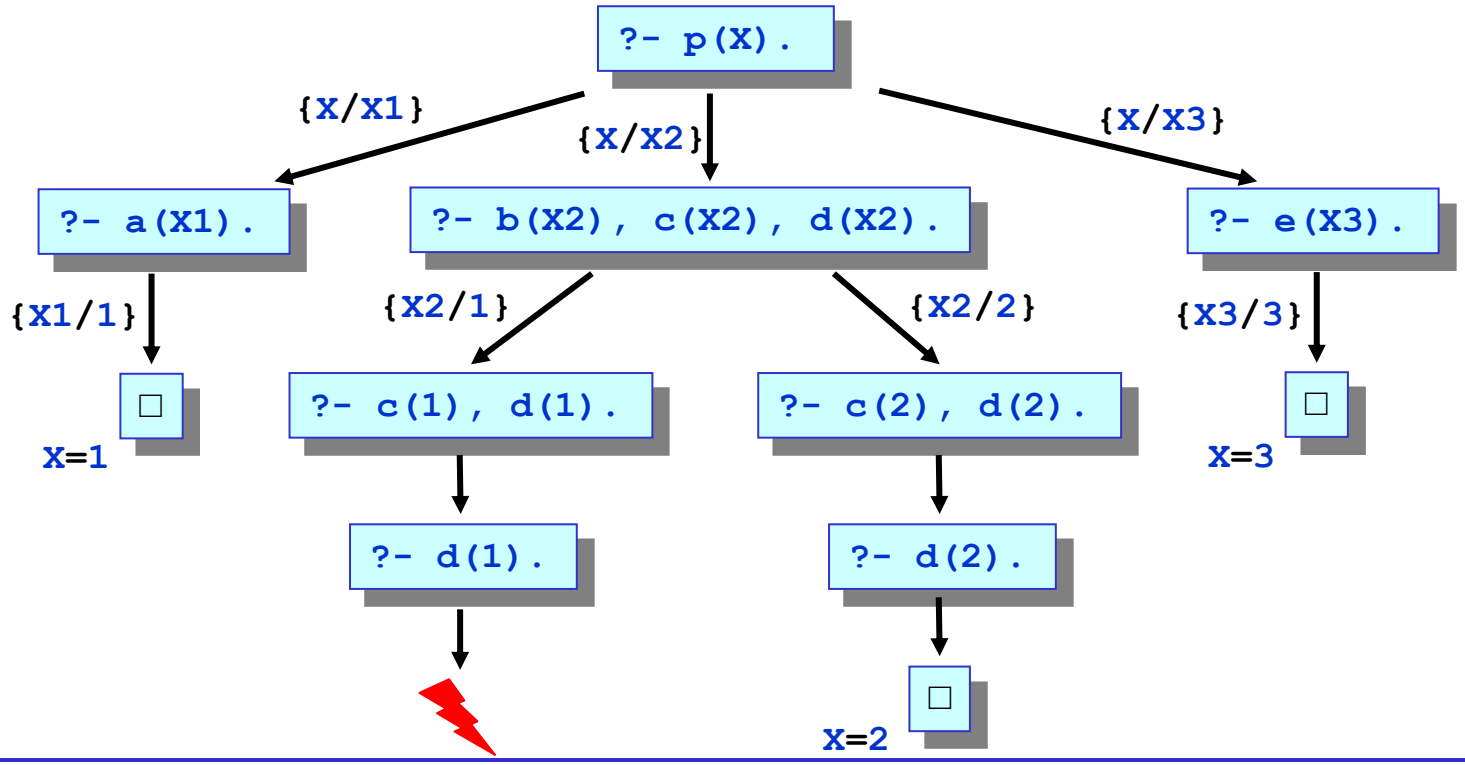
Der „Cut“-Operator (3)

- Allgemein:
  - Der Cut selbst als Teilziel ist immer erfolgreich/erfüllt.
  - Wenn ein anderes Teilziel eine Regel mit Cut benutzt, und dieser Cut im Verlauf der Ableitung erreicht wird, dann sind alle gemachten Entscheidungen seit (und inklusive) Wahl der entsprechenden Regel unumkehrbar.
  - Entscheidungen die nach Antreffen des Cut anstehen, werden jedoch mit all ihren Alternativen untersucht.
  - Und wenn es zum ursprünglichen Teilziel noch Alternativen gab, dann werden diese ebenfalls weiter untersucht.

Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

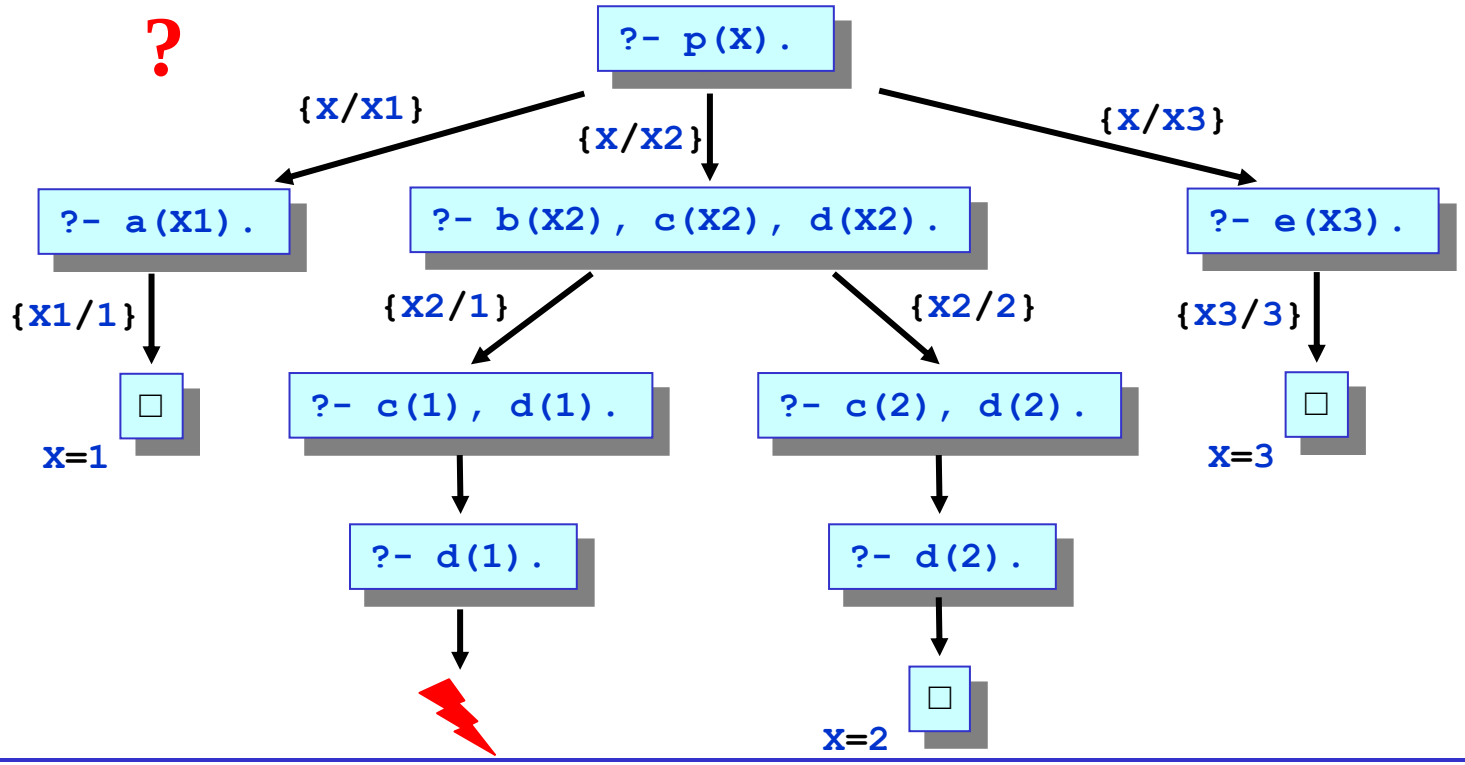
```
p(X) :- a(X).
p(X) :- b(X), c(X), d(X).
p(X) :- e(X).

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



Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

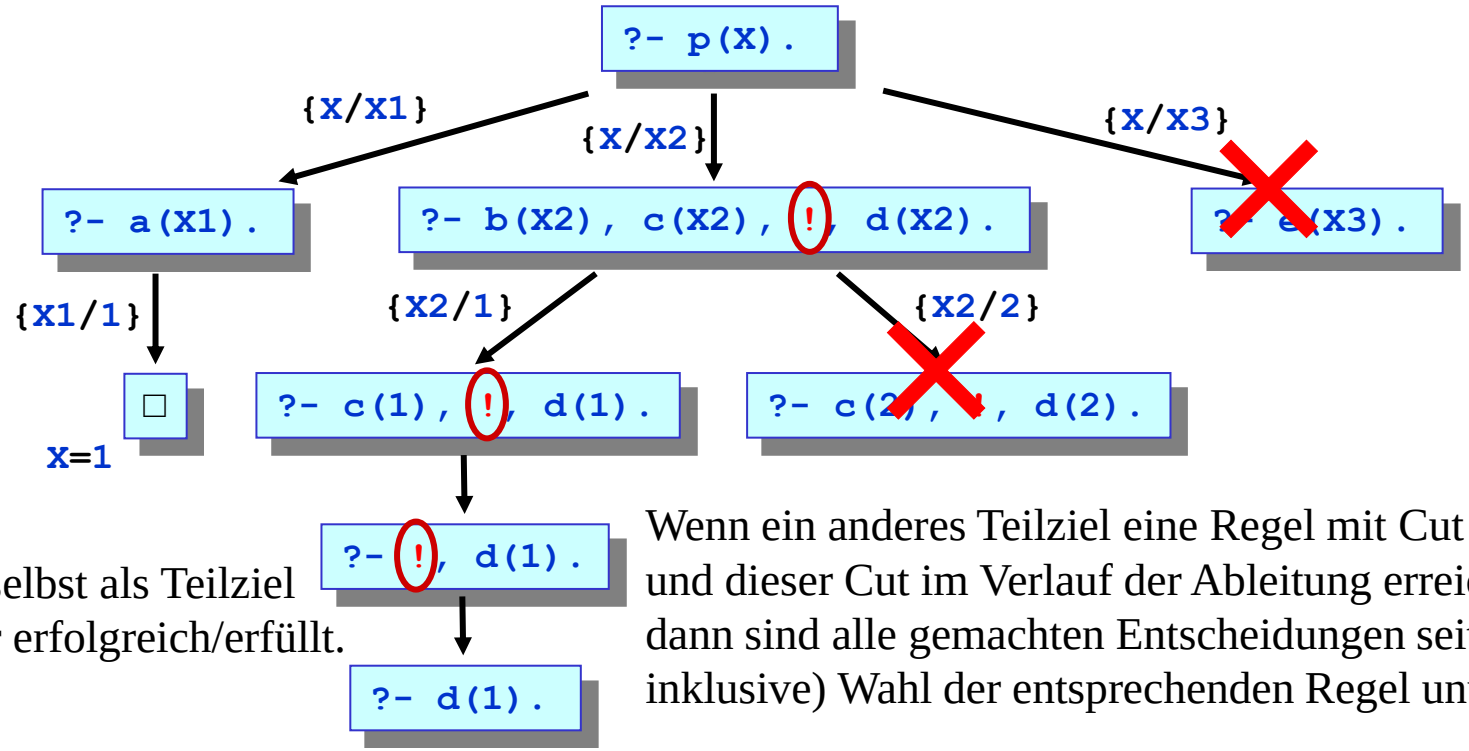
```
p(X) :- a(X).
p(X) :- b(X), c(X), !, d(X).
p(X) :- e(X).

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



Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

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

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



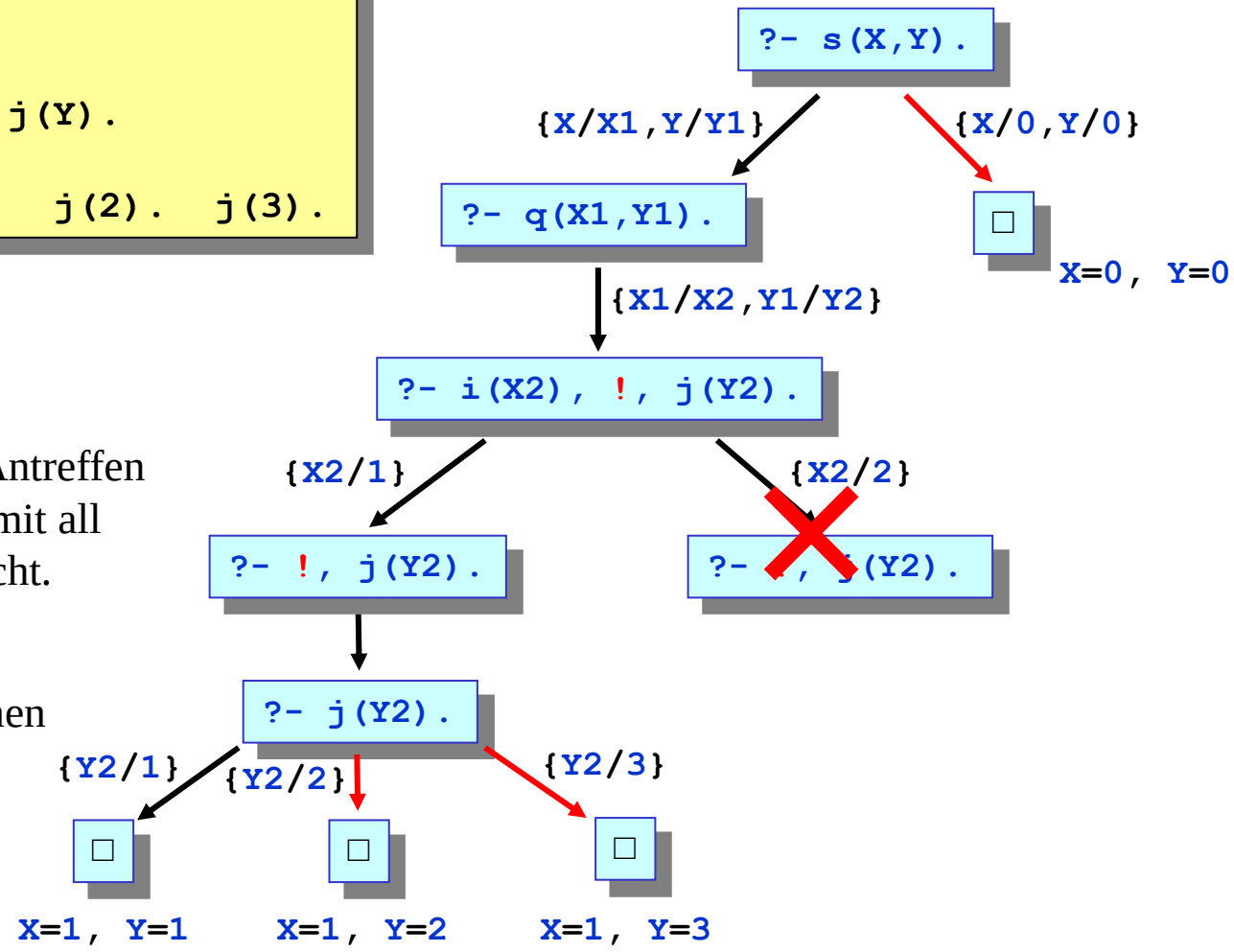
Der Cut selbst als Teilziel ist immer erfolgreich/erfüllt.

Wenn ein anderes Teilziel eine Regel mit Cut benutzt, und dieser Cut im Verlauf der Ableitung erreicht wird, dann sind alle gemachten Entscheidungen seit (und inklusive) Wahl der entsprechenden Regel unumkehrbar.

Der „Cut“-Operator (5)

Illustration an Hand von Ableitungsbäumen:

```
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).
```



Entscheidungen die nach Antreffen des Cut anstehen, werden mit all ihren Alternativen untersucht.

Und wenn es zum ursprünglichen Teilziel noch Alternativen gab, dann werden diese ebenfalls weiter untersucht.

Der „Cut“-Operator (6)

```
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.
```

vs.

```
?- s(2,1).
true.

?- s(2,2).
true.

?- s(2,3).
true.
```

Deklarative Semantik ohne Cut:

$T_P(\emptyset) = \{\texttt{s(0,0)}, \texttt{i(1)}, \texttt{i(2)}, \texttt{j(1)}, \texttt{j(2)}, \texttt{j(3)}\}$

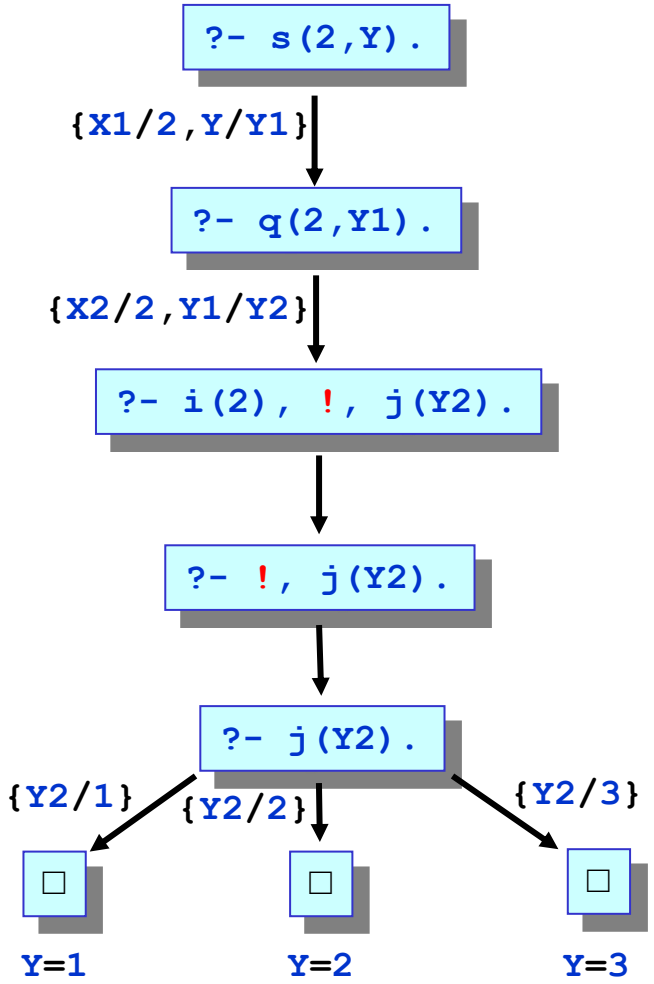
$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\texttt{q(1,1)}, \texttt{q(1,2)}, \texttt{q(1,3)}, \texttt{q(2,1)}, \texttt{q(2,2)}, \texttt{q(2,3)}\}$

$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\texttt{s(1,1)}, \texttt{s(1,2)}, \texttt{s(1,3)}, \texttt{s(2,1)}, \texttt{s(2,2)}, \texttt{s(2,3)}\}$

Der „Cut“-Operator (7)

```
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(2,1).
true.

?- s(2,2).
true.

?- s(2,3).
true.
```



Der „Cut“-Operator – Sinnvolle Verwendung (1)

- Der Cut-Operator kann verwendet werden, um „unnötige“ Vergleiche abzukürzen.
- Zum Beispiel, in Haskell:

```
nodups  [ ]                    =  [ ]
nodups  (x : xs)  | elem x  xs =  nodups xs
                  | otherwise  =  x : nodups xs
```

```
nodups([],[]).
nodups([X|Xs],Ys)       :- member(X,Xs), nodups(Xs,Ys).
nodups([X|Xs],[X|Ys]) :- not(member(X,Xs)), nodups(Xs,Ys).
```

Effizienter:

```
nodups([],[]).
nodups([X|Xs],Ys)       :- member(X,Xs), !, nodups(Xs,Ys).
nodups([X|Xs],[X|Ys]) :- nodups(Xs,Ys).
```

Der „Cut“-Operator – Sinnvolle Verwendung (2)

- Der Cut-Operator kann verwendet werden, um „unnötige“ Vergleiche abzukürzen:

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

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


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

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

- Für bestimmte Aufrufe verhält sich die linke Variante sehr unglücklich. (Für welche?)

Der „Cut“-Operator – Bedingte Anweisung

- Allgemein kann der Cut-Operator verwendet werden, um „bedingte Anweisungen“ zu realisieren:

```
p :- q, !, s.
p :- r.
```

kann interpretiert werden als:

```
if  q  then  s  else  r
```

- Prolog bietet für diese Art Fallunterscheidung eine abkürzende Notation an:

```
p :- q -> s ; r.
```

Jeder der Zweige kann Backtracking durchführen, jedoch nicht der Test selbst (nach erstem Erfülltsein)!

Beispiel:

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

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

Negation und „Cut“

Der „Cut“-Operator kann zur Implementierung der Negation verwendet werden:

```
not(X) :- call(X), !, fail.
not(X).
```

- `call(X)` ist beweisbar, wenn `X` mit einem Term instantiiert ist, der in einer Anfrage vorkommen darf, und die Anfrage `?- X.` beweisbar ist.

ein Metaprädikat (nimmt anderes Prädikat/Literal/Anfrage als Argument)

- `fail` ist ein vordefiniertes Prädikat, für das keine Klauseln existieren, und das daher nie beweisbar ist.

Zusammenfassung Cut

Vorteile Cut:

- kann Effizienz von Programmen erhöhen
- kann Formulierung von Prädikaten vereinfachen

Nachteile Cut:

- lässt sich nur operationell, nicht deklarativ, verstehen
- eine Hauptquelle von Fehlern in Prolog-Programmen (wegen fehlender und/oder falscher Antworten)

Deskriptive Programmierung

Ein- und Ausgabe in Prolog

Ein kleines „interaktives“ Programm (1)

Zugleich mal ein Beispiel für ein etwas „praktischeres“ Prolog-Programm …

Ein kleines numerisches Spiel:

- es gibt zwei Spieler: A und B
- zu Beginn liegen Karten mit den Zahlen 1 bis 9 offen bereit
- die Spieler wählen abwechselnd eine Karte
- gewonnen hat, wer zuerst drei Karten besitzt, deren Summe 15 ergibt

Beispiel … („live“)

Ein kleines „interaktives“ Programm (2)

Ein Hilfsprädikat:

```
move(Xs,X,Ys) :- member(X,Xs), delete(Xs,X,Ys).
```

```
?- move([1,2,3,4,5,6,7,8,9],5,Ys).
Ys = [1,2,3,4,6,7,8,9] ;
false.
```

„Hauptschleife“:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  print((Bs,A1,Ys)), nl,
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```

Benutzt (genaue Definition nicht so wesentlich, hier jetzt nicht besonders elegant):

```
sum15(As) :- move(As,A1,Xs), move(Xs,A2,Ys), move(Ys,A3,_),
             A is A1+A2+A3, A=15.
```

Ein kleines „interaktives“ Programm (3)

Etwas besser „verpackt“:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```

```
start :- As=[], Bs=[], Xs=[1,2,3,4,5,6,7,8,9],
         show(As,Bs,Xs), play(As,Bs,Xs).

show(As,Bs,Xs) :- print((As,Bs,Xs)), nl.
```

Nun würden wir gern gegen einen „intelligenten“ Gegner spielen!

Idee: ein Prädikat **`to_win(As,Bs,Xs,X)`** schreiben, das nächsten Zug ermittelt.

Versuch intelligenten Spielens

Ein optimaler Zug für A siegt entweder sofort oder erzwingt Niederlage von B:

```
to_win(As,Bs,Xs,X) :- move(Xs,X,Ys), A1 = [X|As],
                      (sum15(A1); will_lose(Bs,A1,Ys)).
```

Eine Situation ist aussichtslos für B, wenn kein Unentschieden erreicht, und kein Zug zum direkten Sieg von B führt oder zumindest zu einer Situation, in der A nicht gewinnen kann:

```
will_lose(Bs,As,Ys) :- Ys\=[], not((move(Ys,Y,Zs), B1=[Y|Bs],
                                    (sum15(B1); not(to_win(As,B1,Zs,_)))
                                   )).
```

```
?- to_win([3,9],[4,8],[1,2,5,6,7],X).
X = 5 ;
false.

?- to_win([],[],[1,2,3,4,5,6,7,8,9],A).
false.

?- move([1,2,3,4,5,6,7,8,9],A,Xs),to_win([],[A],Xs,B).
false.
```