CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Logikorientierte Programmierung

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Prolog
- Closed World Assumption
- Resolution und Unifikation
- Relationen
- Constraint Logic Programming over Finite Domains

CARL VON OSSIETZKY universität OLDENBURG

Logikorientierte Programmierung

CARL VON OSSIETZKY universität OLDENBURG

Logikorientierte Programmierung

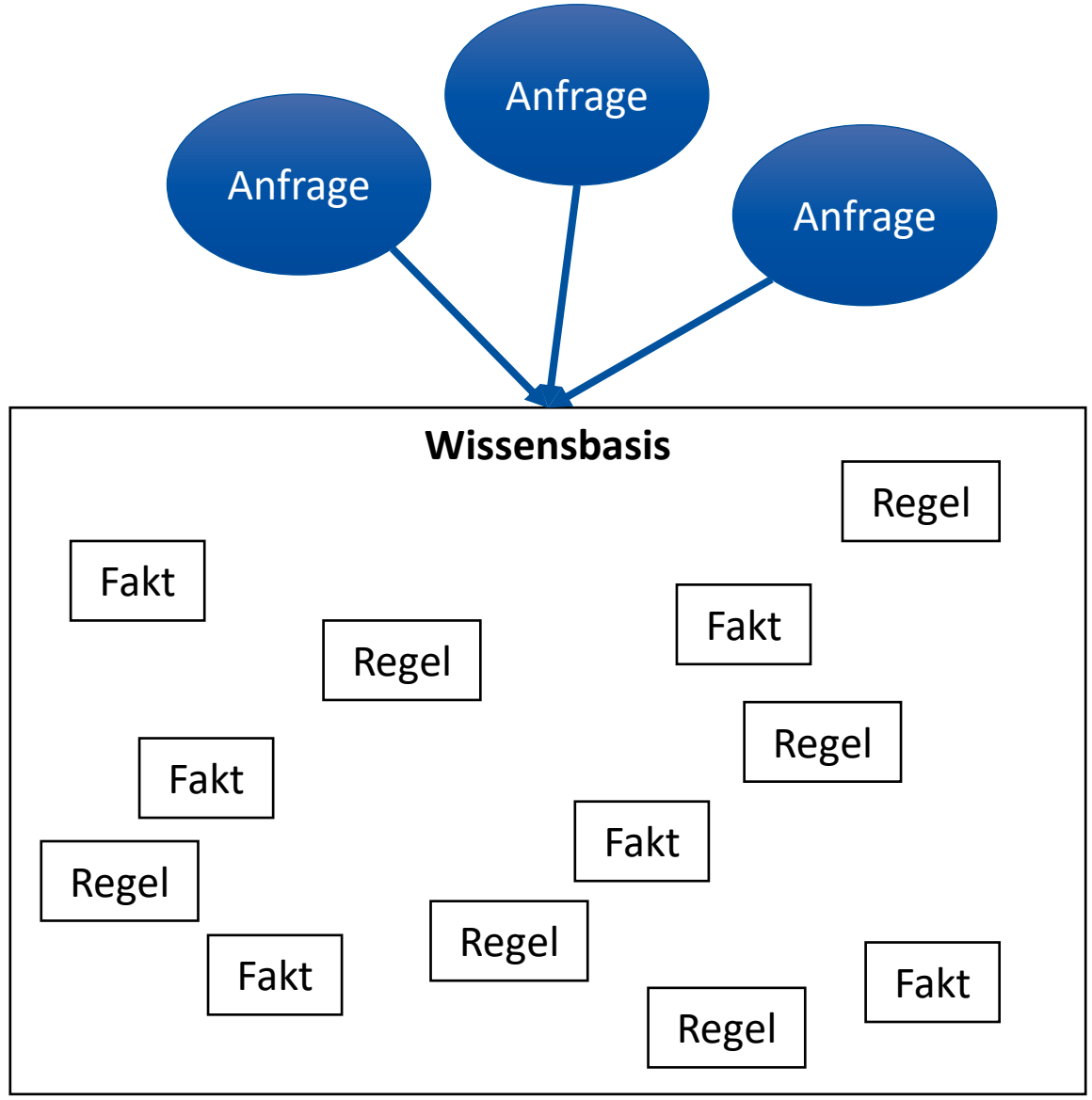
- Basiert auf Aussagen- bzw. Prädikatenlogik
- Ein Programm ist eine Folge logischer Ausdrücke
- Keine Funktionen sondern Relationen
- Das System führt logische Ableitungen durch
  - Wenn es kalt ist und regnet,
    dann brauche ich eine Regenjacke
  - kalt ∧ regnet ⇒ Regenjacke
  - um „Regenjacke“ zu beweisen, zeige „kalt“ und „regnet“
- Programmiersprache Prolog
  - **Pro**gramming in **Log**ic
  - SWI-Prolog
    - http://www.swi-prolog.org/

CARL VON OSSIETZKY universität OLDENBURG

Anwendung von Logikprogrammierung

- Künstliche Intelligenz
- Expertensysteme
- Relationale Datenbanken
  - Datalog
- Theorembeweiser
- Datenanalyse

Grundprinzip



Fakten, Regeln und Anfragen

- Fakten und Regeln formalisieren bekannte Aussagen und vorhandenes Wissen
- Anfragen sind aufgebaut wie Fakten, können aber Platzhalter enthalten
- Beispiele
  - **Fakten**
    - es regnet
    - die Sonne scheint
    - Frodo ist ein Hobbit
    - Bilbo besitzt einen Ring
    - Samwise ist ein Freund von Frodo
  - **Regeln**
    - Hobbits sind klein und hungrig
    - Wer hungrig ist, der isst alles
  - **Anfragen**
    - ***Wer*** ist alles ein Hobbit?
    - Besitzt Frodo ***etwas***?
    - ***Wer*** besitzt einen Ring?

Closed World Assumption

- Die Faktenbasis enthält alles, was wahr ist
- Alles, was nicht in der Faktenbasis steht und auch nicht daraus hergeleitet werden kann, ist falsch
- Beispiel
  - Frodo isst gerne Kartoffeln (Fakt)
  - Isst Frodo gerne Grünkohl? (Anfrage) → nein
    - ➢ Wissensbasis muss (annähernd) vollständig sein!
    - ➢ Closed World Assumption
- Anderer Ansatz: Open World Assumption
  - alles, was nicht klar bestätigt oder negiert ist, und was nicht hergeleitet werden kann, ist unbekannt
  - ➢ Beschreibungslogik

CARL VON OSSIETZKY universität OLDENBURG

Prolog

Syntax

- Fakten
  - `es_regnet.`
  - `'die Sonne scheint'.`
  - `hobbit(frodo).`
  - `besitzt(bilbo, ring).`
  - `freund(samwise, frodo).`
- Regeln
  - `hobbit(X) :- klein(X), hungrig(X).`
    `% wenn X klein und hungrig ist,`
    `% dann ist X ein Hobbit`
  - `isst(X, _) :- hungrig(X).`
    `% wenn X hungrig ist, dann isst X alles`
- Anfragen
  - `hobbit(X).` `% wer ist alles ein Hobbit?`
  - `besitzt(frodo, _).` `% besitzt Frodo etwas?`
  - `besitzt(X, ring).` `% wer besitzt einen Ring?`

`,`	= und
`;`	= oder
`\+`	= nicht
`:-`	= Implikation <-

CARL VON OSSIETZKY universität OLDENBURG

Beispiel

```
hobbit(frodo).
besitzt(bilbo, ring).
freund(samwise, frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).
isst(X, _) :- hungrig(X).
```

```
?- hobbit(X).
X = frodo ;
X = samwise.
?- besitzt(frodo, _).
false.
?- besitzt(X, ring).
X = bilbo.
?- hobbit(samwise).
true.
?- isst(samwise, kaease).
true.
?- freund(frodo, samwise).
false.
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel (2)

```
man(bob).
woman(alice).
man(charlie).
woman(daisy).
% Alice und Bob sind die Eltern von Charlie und Daisy
parents(charlie, alice, bob).
parents(daisy, alice, bob).
% X ist Bruder von Y, wenn X ein Mann ist und die gleichen Eltern hat
brother(X, Y) :- man(X), parents(X, A, B), parents(Y, A, B), X \== Y.
brother(X, Y) :- man(X), parents(X, A, B), parents(Y, B, A), X \== Y.
```

```
?- brother(charlie, daisy).
true.

?- brother(daisy, charlie).
false.

?- brother(X, Y).
X = charlie,
Y = daisy ;
false.
```

CARL VON OSSIETZKY universität OLDENBURG

Resolution und Unifikation

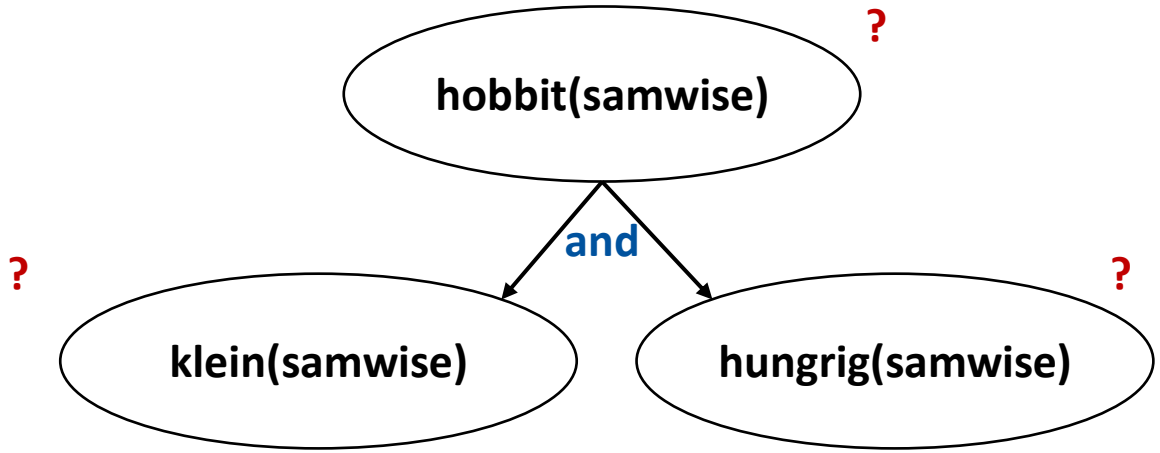
- Auswertung von Anfragen
  - Suchen von direkten Treffern in der Wissensbasis (Matching)
    - `?- hobbit(frodo).`
  - Mapping von Variablen auf Fakten (Unification)
    - `?- hobbit(X).`
  - Weitere Auflösung und Nachverfolgung von Regeln (Resolution)
    - `?- hobbit(samwise).`

Backtracking

```
hobbit(frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(samwise).
true.
```

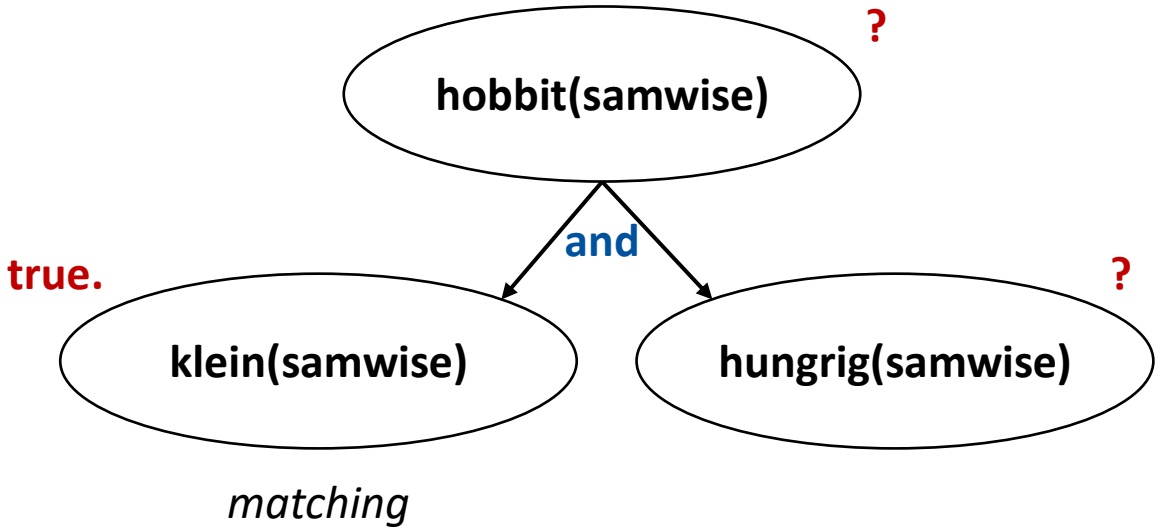
Backtracking

```
hobbit(frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(samwise).
true.
```



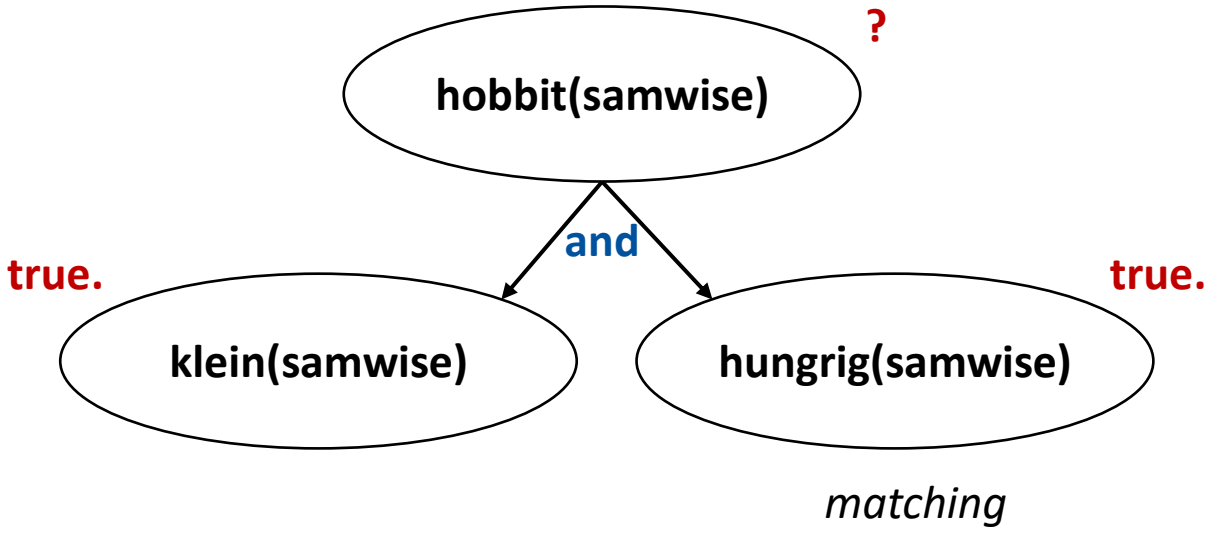
Backtracking

```
hobbit(frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(samwise).
true.
```



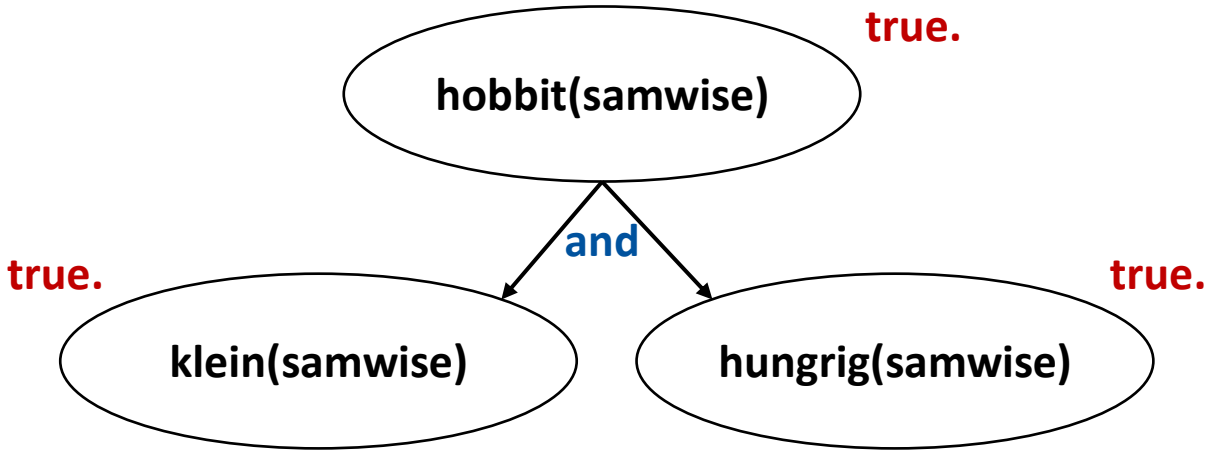
Backtracking

```
hobbit(frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(samwise).
true.
```



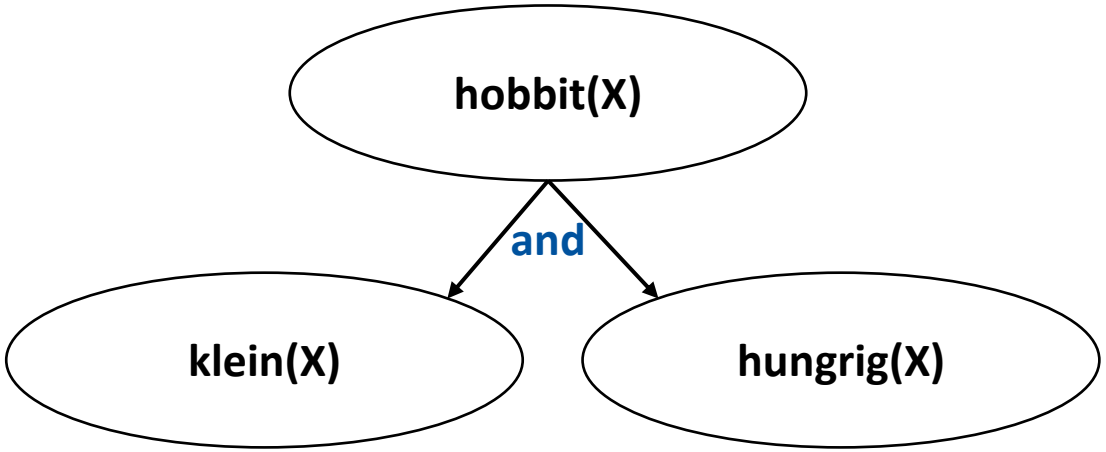
Backtracking

```
hobbit(frodo).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(samwise).
true.
```



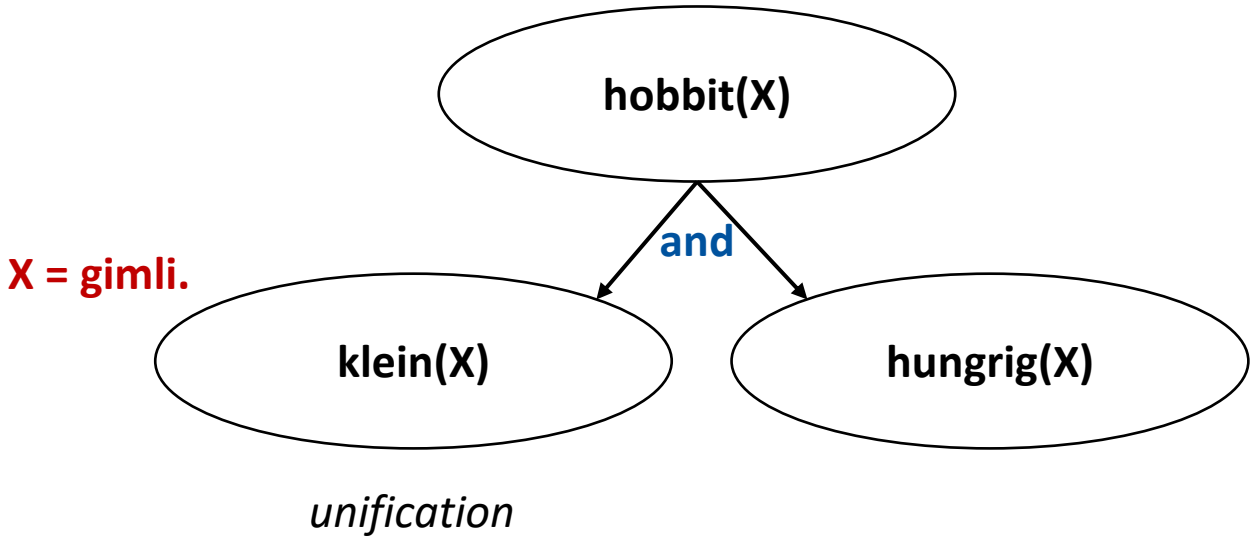
Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



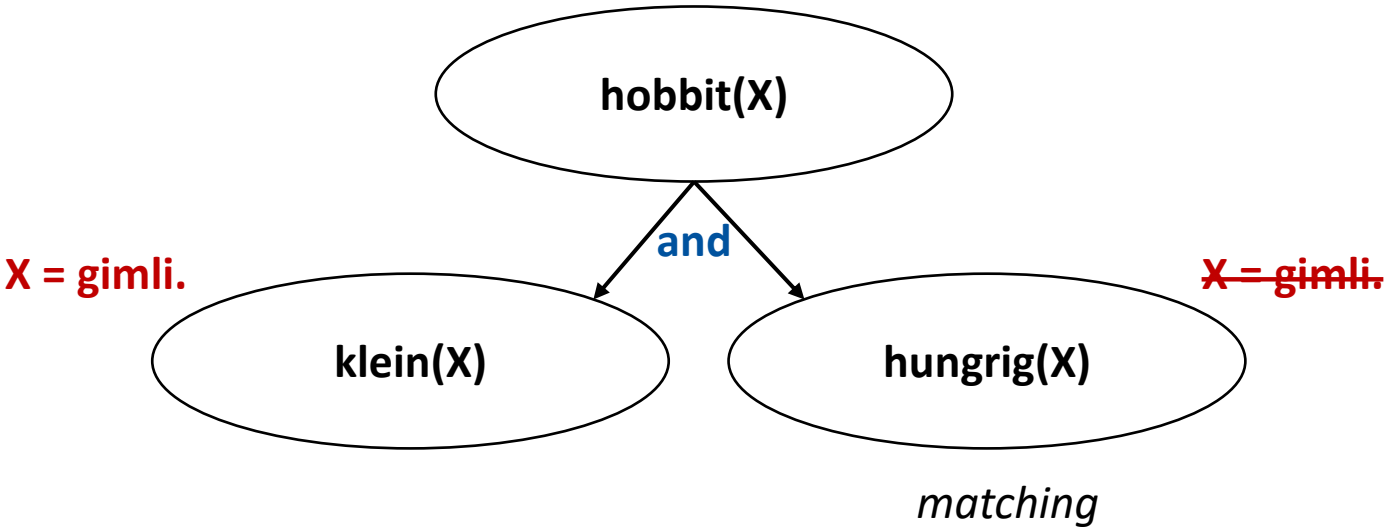
Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



Backtracking (2)

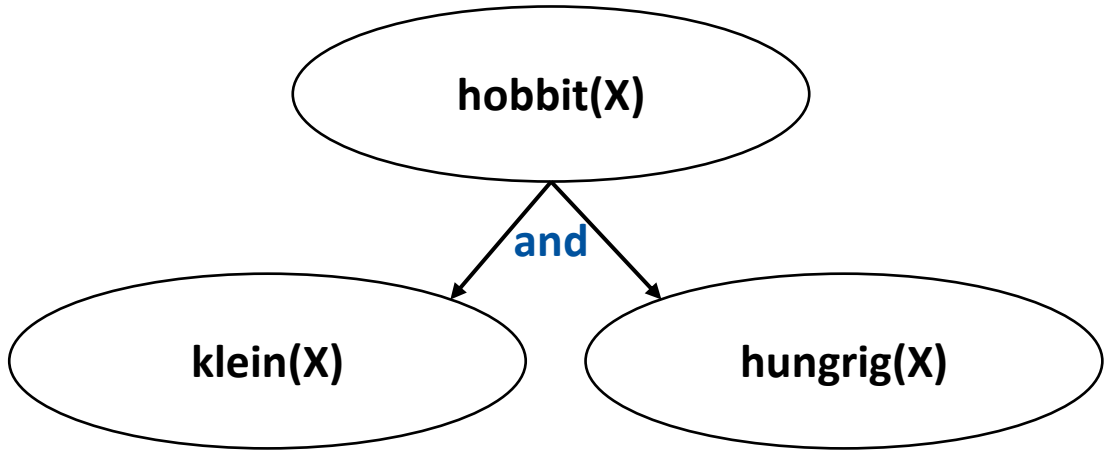
```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```

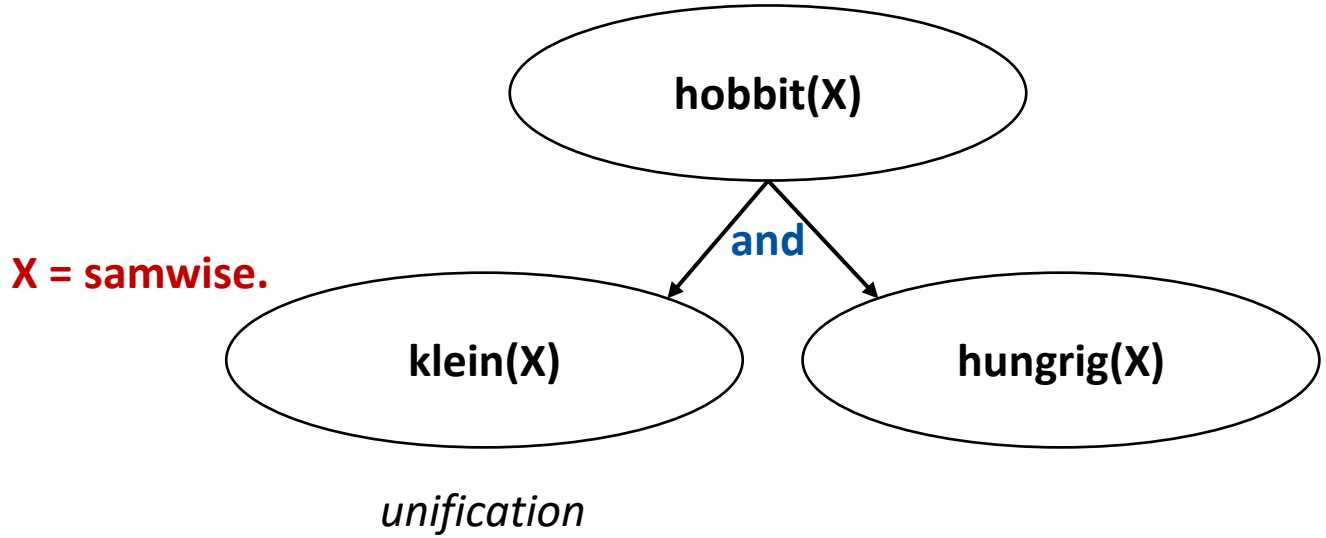
X =



backtracking

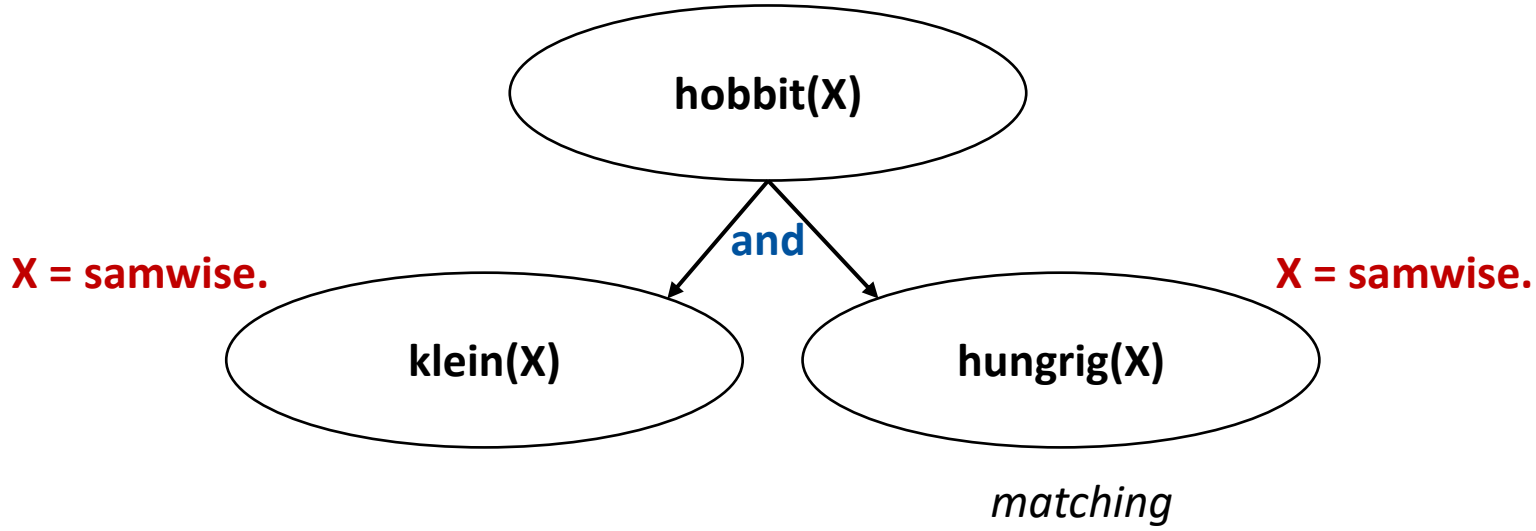
Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



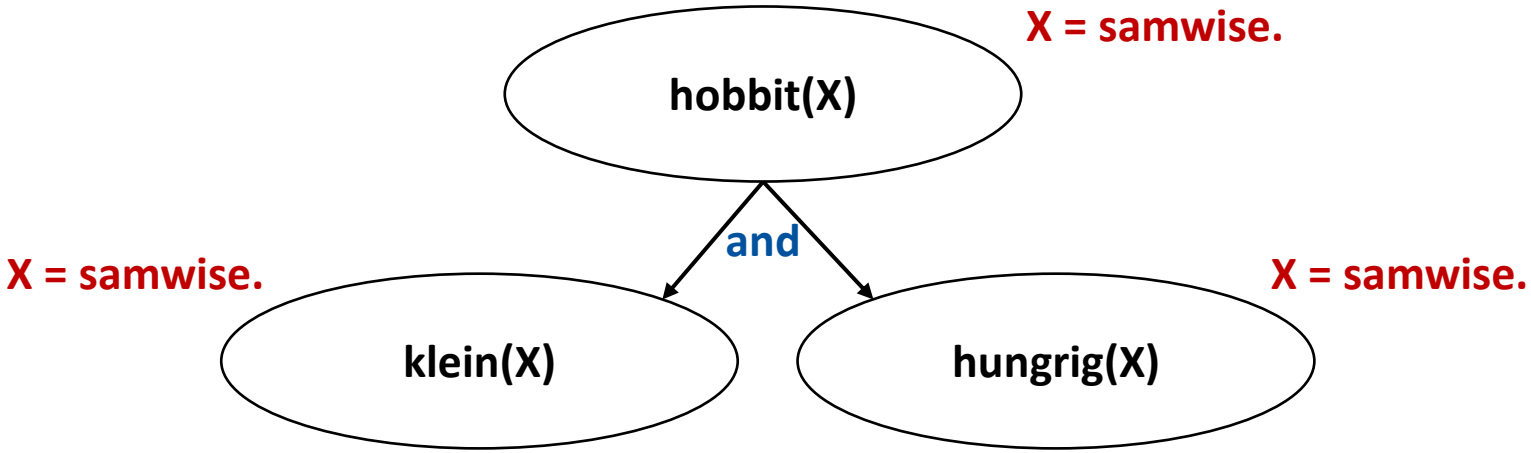
Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



Backtracking (2)

```
klein(gimli).
hungrig(samwise).
klein(samwise).
hobbit(X) :- klein(X), hungrig(X).

?- hobbit(X).
X = samwise.
```



Listen

- Ähnliche Syntax wie SML
  - `[1, 2, 3, 4]`
  - `[[a, b], [c, d]]`
  - `[X|L]` *% in SML: x::l*

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

Alles, was nicht explizit definiert ist, ist falsch
→ **hier** kein Basisfall nötig!

Dafür ist *pattern-matching* statt des *oder*-Ausdrucks möglich
(Variable **X** kann mehrfach auftauchen).

```
fun member(x, nil)    = false
  | member(x, y::ys) = x=y orelse member(x, ys);
```

Relationen

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

Anders lesen als in SML (Relationen statt Funktionen):
Nicht „Das Ergebnis von member von X und L ist der rekursive Aufruf…“ (2. Zeile)
Sondern
„X und eine beliebige Liste, die mit X beginnt,
stehen immer in einer member-Beziehung“ (1. Zeile)
„X und eine Liste L, die mit irgendwas beginnt, stehen dann in einer member-Beziehung,
wenn X und L in einer member-Beziehung stehen“ (2. Zeile)
→ keine Unterscheidung von Eingabe und Ausgabe/Ergebnis

```
1. append([], L, L).
2. append([J|K], L, [J|KL]) :- append(K, L, KL).
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Relationen

```
append([], L, L).
append([J|K], L, [J|KL]) :- append(K, L, KL).
```

```
?- append([1, 2], [3, 4], X).
X = [1, 2, 3, 4].
```

```
?- append([1, 2], X, [1, 2, 3, 4]).
X = [3, 4].
```

CARL VON OSSIETZKY universität OLDENBURG

CLP(FD)

- Constraint Logic Programming over Finite Domains
  - constraint: Einschränkung, Bedingung eines Optimierungsproblems
  - constraint programming: Beziehungen zwischen Variablen werden über constraints ausgedrückt
  - logic programming: Programmierung mit formaler Logik
  - constraint logic programming: Programmierung mit formaler Logik, um die Erfüllung von constraints zu bestimmen
  - domain: Wertebereich
  - finite domain: Endlicher Wertebereich
- CLP(FD) besteht oft aus zwei Phasen
  1. der Beschreibung des Problems als Liste von constraints (Modell)
  2. der Suche nach Lösungen, die das Modell erfüllen

```
?- use_module(library(clpfd)).
```

CARL VON OSSIETZKY universität OLDENBURG

Arithmetik

```
?- X is 3 + 4.
X = 7.
```

```
?- 7 is X + 4.
ERROR: Arguments are not sufficiently instantiated
```

```
?- X #= 3 + 4.
X = 7.
```

```
?- 7 #= X + 4.
X = 3.
```

```
?- X #= 2^4.
X = 16.
```

```
?- 16 #= 2^X.
X = 4.
```

Beispiel: n-Damen-Problem - Problembeschreibung

```
n_queens(N, Qs) :-
    length(Qs, N),
    Qs ins 1..N,
    safe_queens(Qs).
```

N: Anzahl Damen, Qs: Liste mit Spalten-Positionen
Zusammenhang zwischen N und Qs: Qs hat Länge N
Wertebereich von Qs: Qs enthält nur Werte zwischen 1 und N
} Feld hat Größe N×N
constraint: alle Positionen in Qs müssen sicher sein

```
safe_queens([]).
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).
```

```
safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
    Q0 #\= Q,
    abs(Q0 - Q) #\= D0,
    D1 #= D0 + 1,
    safe_queens(Qs, Q0, D1).
```

Prüft die Sicherheit von Q0 bzgl. Qs durch Abgleich der Spalten und der Diagonalen D0, D0+1, D0+1+1, …

CARL VON OSSIETZKY universität OLDENBURG

Labeling

```
?- length(L,5), L ins 1..8, label(L).
L = [1, 1, 1, 1, 1] ;
L = [1, 1, 1, 1, 2] ;
L = [1, 1, 1, 1, 3] ;
L = [1, 1, 1, 1, 4] ;
L = [1, 1, 1, 1, 5] ;
L = [1, 1, 1, 1, 6] ;
L = [1, 1, 1, 1, 7] ;
L = [1, 1, 1, 1, 8] ;
L = [1, 1, 1, 2, 1] ;
L = [1, 1, 1, 2, 2] ;
L = [1, 1, 1, 2, 3] ;
L = [1, 1, 1, 2, 4] .
```

`label(L)` erzeugt eine Permutation der möglichen Werte einer Liste von Variablen **`L`**.

`label(L)` ist äquivalent zu **`labeling([], L)`**.

CARL VON OSSIETZKY universität OLDENBURG

Labeling

`labeling([], L)` erhält als ersten Parameter eine Liste von Auswahlstrategien.

```
?- length(L,5), L ins 1..8, labeling([ff], L).
L = [1, 1, 1, 1, 1] ;
L = [1, 1, 1, 1, 2] ;
L = [1, 1, 1, 1, 3] .
```

```
?- length(L,5), L ins 1..8, labeling([down], L).
L = [8, 8, 8, 8, 8] ;
L = [8, 8, 8, 8, 7] ;
L = [8, 8, 8, 8, 6] .
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: n-Damen-Problem

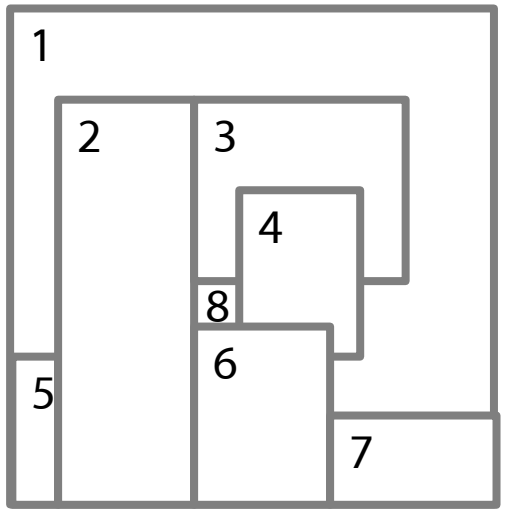
```
?- n_queens(8, Qs), label(Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] ;
Qs = [1, 6, 8, 3, 7, 4, 2, 5] ;
Qs = [1, 7, 4, 6, 8, 2, 5, 3] ;
Qs = [1, 7, 5, 8, 2, 4, 6, 3] .
```

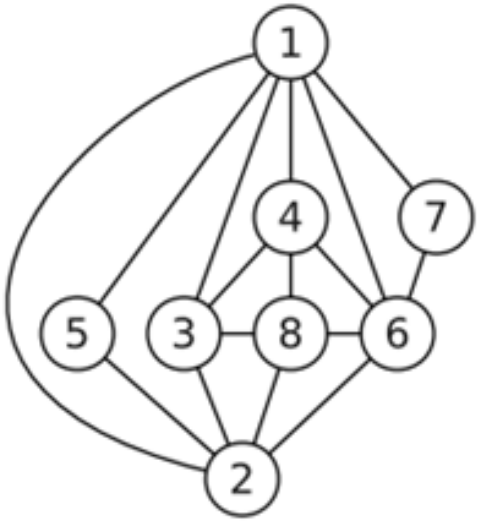
```
?- n_queens(8, Qs), labeling([ff], Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] ;
Qs = [1, 6, 8, 3, 7, 4, 2, 5] ;
Qs = [1, 7, 4, 6, 8, 2, 5, 3] ;
Qs = [1, 7, 5, 8, 2, 4, 6, 3] .
```

```
?- n_queens(80, Qs), labeling([ff], Qs).
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 68|...] ;
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 76|...] ;
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 70|...] ;
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 77|...] .
```

Beispiel: Karten Färben

- Aufgabe: Färbe eine Landkarte mit einer minimalen Anzahl an Farben, so dass zwei Nachbargebiete nie die gleiche Farbe haben





CARL VON OSSIETZKY universität OLDENBURG

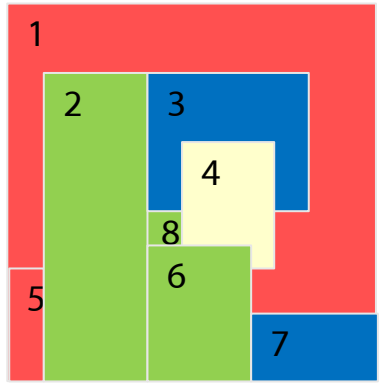
Adjazenzliste

```
map([[1,2],[1,3],[1,4],[1,5],[1,6],[1,7],
     [2,3],[2,5],[2,6],[2,8],[3,4],[3,8],
     [4,6],[4,8],[6,7],[6,8]]).
```

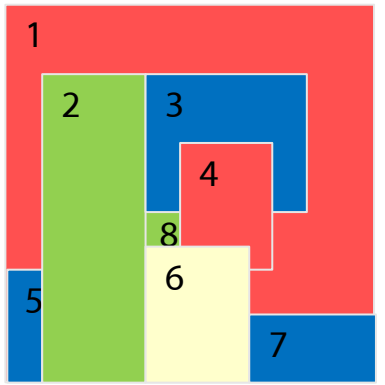
```
adjacent(X, Y, Map) :-
    member([X, Y], Map);
    member([Y, X], Map).
```

Färbeversuche

```
col1([[1,red],[2,green],[3,blue],
      [4,yellow],[5,red],[6,green],
      [7,blue],[8,green]]).
```

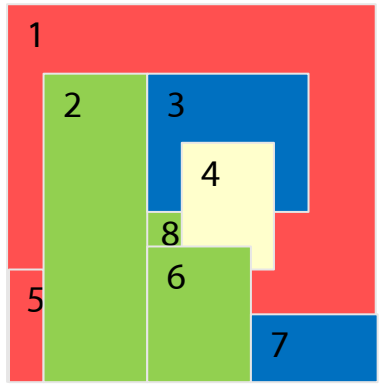



```
col2([[1,red],[2,green],[3,blue],
      [4,red],[5,blue],[6,yellow],
      [7,blue],[8,green]]).
```



Konflikte

```
conflict(Map, Coloring) :-
    member([X, Color], Coloring),
    member([Y, Color], Coloring),
    adjacent(X, Y, Map).
```




```
conflict([[1,2],[1,3],[1,4],[1,5],
[1,6],[1,7],[2,3],[2,5],[2,6],[2,8
],[3,4],[3,8],[4,6],[4,8],[6,7],[6
,8]],[[1,red],[2,green],[3,blue],[
4,yellow],[5,red],[6,green],[7,blu
e],[8,green]]).
yes.
```

CARL VON OSSIETZKY universität OLDENBURG

Duplikatenfreie Liste der Regionen

```
regions([], R, R).
regions([[X, Y]|S], R, A) :- (
    member(X, R) -> (
        member(Y, R) -> regions(S, R, A);
                        regions(S, [Y|R], A)
    ); (
        member(Y, R) -> regions(S, [X|R], A);
                        regions(S, [X,Y|R], A)
    )
).
```

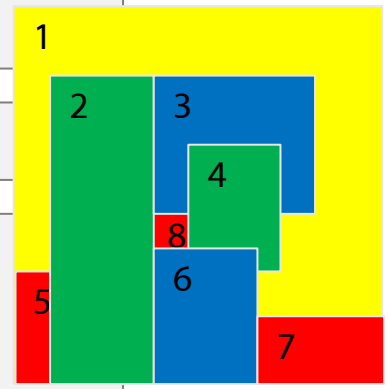
```
?-
regions([[1,2],[1,3],[1,4],[1,5],[1,6],[1,7],[
2,3],[2,5],[2,6],[2,8],[3,4],[3,8],[4,6],[4,8]
,[6,7],[6,8]], [], R).
R = [8, 7, 6, 5, 4, 3, 1, 2].
```

Färbungen Erzeugen

```
% color_all(Regions, Colors, Coloring)
% erzeugt uneingeschraenkt alle moeglichen Faerbungen
color_all([], _, []).
color_all([R|Rs], Colors, [[R,C]|A]) :-
    member(C, Colors),
    color_all(Rs, Colors, A).

% erzeugt alle gueltigen Faerbungen
color(Map, Colors, Coloring) :-
    regions(Map, [], Regions),
    color_all(Regions, Colors, Coloring),
    \+ conflict(Map, Coloring).
```

```
colors([red,blue,green,yellow]).
```

```
?- map(M),colors(C),color(M,C,Col).
Col = [[8, red], [7, red], [6, blue], [5, red],
[4, green], [3, blue], [1, yellow], [2, green]]
```



CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Prolog
- Closed World Assumption
- Resolution und Unifikation
- Relationen
- Constraint Logic Programming over Finite Domains