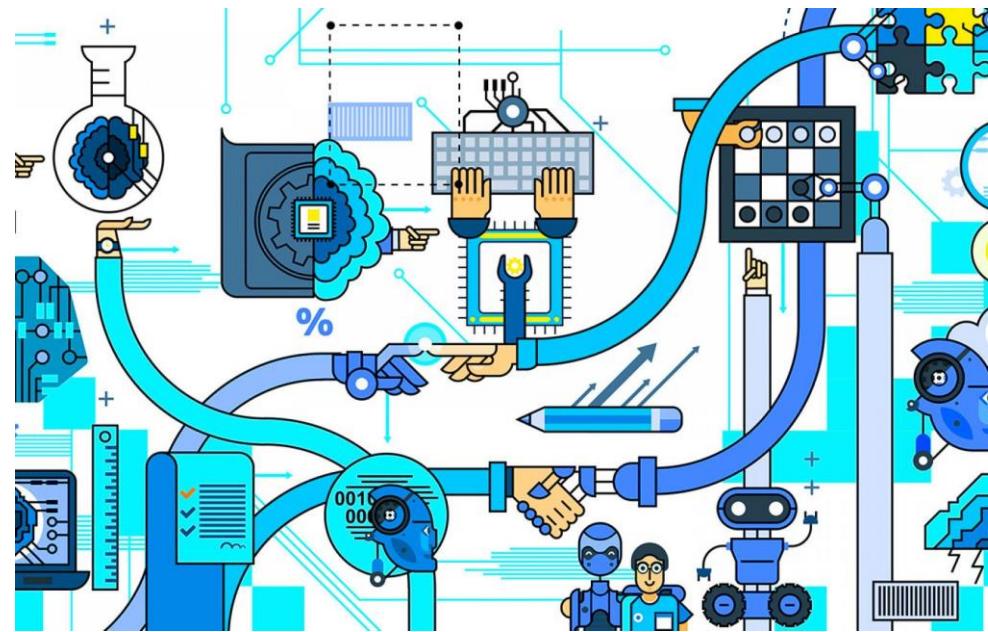


Logische Programmierung

Einführung in PROLOG



Organisatorisches

- In diesem Kurs (Vorlesungen und Übungen) verwenden wir ausschließlich die kostenlose Prolog-Implementierung von SWI.
- Der SWI-Prolog-Interpreter kann unter <http://www.swi-prolog.org> für verschiedene Betriebssysteme (Windows, Mac OS, Linux, Unix) heruntergeladen werden.
- Wir können auch die online Implementation SWISH PROLOG benutzen: <https://swish.swi-prolog.org>



Kenntnisse

- Es werden keine Vorkenntnisse vorausgesetzt!
- Verfügen Sie (noch) über Kenntnisse in Logik (Aussagenlogik, Prädikatenlogik)? Diese sind von Vorteil, um die Grundlagen der logischen Programmierung zu verstehen!



Erfahrung des letzten Jahres:

- Alle die regelmäßig an den Übungssitzung und der Vorlesung teilgenommen haben, haben die Klausur bestanden.
- Installieren Sie sich Prolog zu Hause und üben Sie regelmäßig.
- Programmieren lernt man nur durch Programmieren!



Ziele

In diesem Kurs lernen Sie:

- ein Problem logisch bzw. deklarativ zu lösen.
- einfache Programme in Prolog zu schreiben.
- den Umgang mit dem Prolog-Interpreter von SWI.



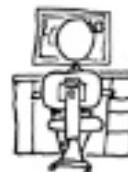
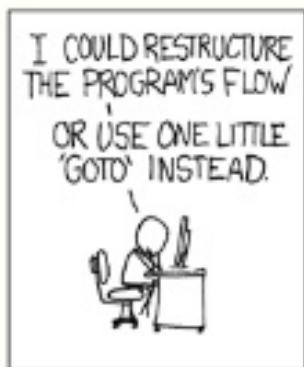
Literatur

Patrick Blackburn, Johan Bos, Kristina Striegnitz (2006).

Learn Prolog Now! (Texts in Computing, Vol. 7). College Publications.
(Online: <http://www.learnprolognow.org/>)



LOGIC PROGRAMMING?



<http://xkcd.com/282/>

Saving developers from imperative velociraptor attacks, one logic program at a time...



Teil 1: Fakten, Regeln und Anfragen

- Prolog (aus dem franz. Programming en Logique) ist eine logische Programmiersprache, die in den 1970er Jahren von dem Informatiker Alain Colmerauer entwickelt wurde.
- Der Ursprung der logischen Programmierung liegt in dem automatisierten Beweisen mathematischer Sätze.
- Die logische Programmierung basiert auf der Syntax der Prädikatenlogik erster Stufe (PL1), die in der zweiten Hälfte des 19. Jahrhunderts von Gottlob Frege aufgestellt wurde.



Anwendungsgebiete

- Entwicklung von Expertensystemen
- Computerlinguistik
- Künstliche Intelligenz
- Die Sprachverarbeitungskomponenten des KI-Systems Watson von IBM wurden in Prolog geschrieben.
- Watson bei Jeopardy (the IBM Challenge):
<https://www.youtube.com/watch?v=P18EdAKuC1U>



Grundprinzip

- Prolog-Programme bestehen aus einer Wissensdatenbank, die aus Fakten und Regeln bestehen.
- Der Benutzer formuliert Anfragen an die Wissensdatenbank.
- Prolog versucht mittels bestehender Fakten und Regeln eine Antwort zu finden.
- Wenn Prolog eine Antwort findet, bedeutet dies, dass die Anfrage logisch ableitbar ist.
- Der Programmier-Ansatz von Prolog folgt dem deklarativen Programmierparadigma.



Programmierparadigma

- Ein Programmierparadigma ist ein fundamentaler Programmierstil/-ansatz.
- Bei der Programmierung wird zwischen zwei grundlegenden Programmierparadigmen unterschieden: Imperative Programmierung und **deklarative Programmierung**
- Einige Sprachen der imperativen Programmierung: prozedural (C, Fortran), Objekt-orientiert (C++, Java, Python)
- Einige Sprachen der deklarativen Programmierung: Abfragesprachen (SQL), funktionale Sprachen (Lisp, Haskell), logische Sprachen (Prolog, ASP)



Vergleich der Programmierparadigmen

Bei imperativen Programmiersprachen (Python) wird beschrieben **WIE** ein Problem gelöst werden soll (algorithmisches Ablauf).

```
def istGluecklich(n):
    if n == "tobi":
        print "true"
    else:
        print "false"
```

- istGluecklich("tobi") true
- istGluecklich("otto") false

Bei deklarativen Programmiersprachen (Prolog) wird beschrieben **WAS** gelöst werden soll (Deklaration der Zusammenhänge).

```
istGluecklich(tobi).
```

```
?- istGluecklich(tobi).
true.
?- istGluecklich(otto).
false.
```

Prolog's Programmierparadigma

- Eine ganz einfache Wissensbasis in Prolog erlaubt uns vier unterschiedliche Anfragen zu stellen:

```
mag(pluto,eis).  
mag(pluto,orangen).  
mag(popeye,eis).
```

① Mag Popeye Eis?

```
mag(popeye,eis).  
true.
```

② Wer mag Eis?

```
mag(X,eis).  
X = pluto;  
X = popeye
```



Prolog's Programmierparadigma

- Eine ganz einfache Wissensbasis in Prolog erlaubt uns vier unterschiedliche Anfragen zu stellen:

```
mag(pluto,eis).  
mag(pluto,orangen).  
mag(popeye,eis).
```

③ Was mag Pluto?

```
mag(pluto,X).  
X = eis;  
X = orangen;
```

④ Wer mag was?

```
mag(X,Y).  
X = pluto,  
Y = eis;  
X = pluto,  
Y = orangen; | X = popeye,  
Y = eis
```



Wie funktioniert Prolog?

- Prolog ist eine deklarative Programmiersprache!
- Für den Umgang mit Prolog ist es zwingend erforderlich sich vom imperativen Programmierparadigma zu lösen.
- Wir müssen verstehen wie Prolog „denkt“!
- Dies ist eine essentielle Grundvoraussetzung für die Programmierung in Prolog.



Der Interpreter

- Der Interpreter von Prolog benötigt Informationen um eine Anfrage (query) zu beantworten.
- Diese Informationen werden in einer Wissensbasis (knowledge base) gespeichert.
- Eine Wissensbasis besteht aus Klauseln (clauses). Klauseln sind entweder Fakten (facts) oder Regeln (rules).

Wissenbasis

Pluto ist ein Hund.

Snoopy ist ein Hund.

Popeye ist ein Seemann.

Popeye mag Spinat.

Popeye ist stark, wenn Popeye Spinat mag.



Der Interpreter

- Der Interpreter von Prolog benötigt Informationen um eine **Anfrage** (query) zu beantworten.
- Diese Informationen werden in einer **Wissensbasis** (knowledge base) gespeichert.
- Eine Wissensbasis besteht aus **Klauseln** (clauses). Klauseln sind entweder **Fakten** (facts) oder **Regeln** (rules).

Wissenbasis

Pluto ist ein Hund.
Snoopy ist ein Hund.
Popeye ist ein Seemann.
Popeye mag Spinat.
Popeye ist stark, wenn Popeye Spinat mag.

Anfragen

Ist Pluto ein Hund?
Mag Pluto Spinat?
Ist Garfield ein Hund?
Ist Popeye ein Mann?
Ist Popeye stark?



Der Interpreter

- Der Prolog-Interpreter wird die erste und letzte Frage **bejahen** und die anderen **verneinen**.
- Für Prolog ist alles ‘wahr’ oder ‘beweisbar’, was als Fakt in der Wissensbasis steht oder sich mithilfe von Regeln in der Wissensbasis aus diesen Fakten herleiten lässt.

Wissenbasis

Pluto ist ein Hund.
Snoopy ist ein Hund.
Popeye ist ein Seemann.
Popeye mag Spinat.
Popeye ist stark, wenn Popeye Spinat mag.

Anfragen

Ist Pluto ein Hund?
Mag Pluto Spinat?
Ist Garfield ein Hund?
Ist Popeye ein Mann?
Ist Popeye stark?



Eine Wissensbasis in Prolog

In Prolog wird eine Wissensbasis wie folgt geschrieben:

```
ist_ein_hund(pluto).  
ist_ein_hund(snoopy).  
ist_ein_seemann(popeye).  
mag(popeye, spinat).  
  
ist_stark(popeye) :- mag(popeye, spinat).
```

Diese Wissensbasis besteht aus vier Fakten und einer Regel. Sie definiert vier unterschiedliche Prädikate (predicates) nämlich `ist_ein_hund/1`, `ist_ein_seemann/1`, `mag_spinat/1` und `ist_stark/1`. Anfragen werden in der Konsole an den Interpreter gestellt und ausgewertet:

```
-? ist_ein_hund(pluto).  
true.
```



Eine Wissensbasis in Prolog

```
ist_ein_hund(pluto).  
ist_ein_hund(snoopy).  
ist_ein_seemann(popeye).  
mag(popeye,spinat).  
  
ist_stark(popeye) :- mag(popeye,spinat).
```

Das Symbol ‘:-’ steht für ‘wenn’ oder ‘folgt aus’. Die Regel
‘ist_stark(popeye) :- mag(popeye,spinat).’ kann gelesen werden als
“‘ist_stark(popeye)’ ist wahr, wenn ‘mag(popeye,spinat)’ wahr ist”.



Fakten, Regeln, Klauseln

```
ist_ein_seemann(popeye).  
mag(popeye,spinat).
```

```
ist_stark(popeye) :- mag(popeye,spinat).  
hat_muskeln(popeye) :- hat_trainiert(popeye).  
hat_muskeln(popeye) :- ist_stark(popeye).
```

- Die linke Seite einer Regel nennt man **Regelkopf** (*head of the rule*) und die rechte Seite **Regelkörper** (*body of the rule*).
- Fakten und Regeln einer Wissensbasis nennt man **Klauseln** (*clauses*).
- Bei einem Fakt kann man auch von einer leeren Regel (einer Regel ohne Regelkörper) sprechen:
`'mag(popeye,spinat).' ist äquivalent zu 'mag(popeye,spinat) :- .'.`



Übung

Drücke folgende Fakten und Regeln in Prolog aus:

1. Lena ist hungrig.
2. Lena ist Ottos Mutter.
3. Lena mag alle, die ihr Schokolade schenken.
4. Lena mag alle, die gut singen können und gut kochen.
5. Lena mag alle, die ihr Schokolade oder Kekse schenken.
6. Alle Menschen sind sterblich.
7. Sokrates ist sterblich.
8. Eine Tochter einer Person ist ein weibliches Kind dieser Person.
9. Alle Hunde mögen Wurst.
10. Pluto mag alles, was Mickey ihm gibt.



Regeln und Inferenzen

```
ist_ein_seemann(popeye).  
mag(popeye,spinat).  
ist_stark(popeye) :- mag(popeye,spinat).  
hat_muskeln(popeye) :- hat_trainiert(popeye).  
hat_muskeln(popeye) :- ist_stark(popeye).
```

- Wenn der Regelkörper wahr ist (sich also aus der Wissensbasis herleiten lässt), so ist auch der Regelkopf wahr.
- Dieses Deduktionsprinzip heißt **Modus Ponens**:

$$\frac{a \rightarrow b \quad a}{b} \qquad \frac{b : - a. \quad mag(popeye, spinat).}{mag(popeye, spinat).} \qquad \frac{mag(popeye, spinat).}{ist_stark(popeye)}.$$

- Aus der Regel '`ist_stark(popeye) :- mag(popeye,spinat).`' und dem Fakt '`mag(popeye,spinat).`' **inferiert** oder **schließt** der Prolog-Interpreter, dass '`ist_stark(popeye).`' gilt.



Stelligkeit

Fakten und Prädikate können beliebige Stelligkeit haben:

```
es_Regnet.  
mag(popeye, spinat).  
mag_spinat(popeye).
```

Der Fakt `es_Regnet`/0 ist nullstellig.
Der Fakt `mag`/2 ist zweistellig.
Der Fakt `mag_spinat`/1 ist einstellig.

Beachte, die beiden Fakten `mag(popeye, spinat)`. `mag_spinat(popeye)`. modellieren zwar denselben Sachverhalt, sie sind aber nicht äquivalent.

0-stellige Fakten werden auch **atomare Fakten** genannt.



Regeln mit mehr als einem Ziel

- Das Komma drückt eine **Konjunktion** aus: Popeye mag Spinatnudeln, wenn Popeye Spinat mag **und** wenn Popeye Nudeln mag.

```
mag(popeye, spinatnudeln) :-  
    mag(popeye, spinat),  
    mag(popeye, nudeln).
```

- Das Semikolon steht für eine **Disjunktion**: Popeye ist stark, wenn Popeye Spinat mag **oder** wenn Popeye trainiert hat.

```
ist_stark(popeye) :-  
    mag(popeye, spinat);  
    hat_trainiert(popeye).
```

Zur Verbesserung der Lesbarkeit sollte man in Prolog besser zwei Regeln statt einer disjunktiven schreiben:

```
ist_stark(popeye) :- mag(popeye, spinat).  
ist_stark(popeye) :- hat_trainiert(popeye).
```



Variablen

```
liebt(pluto,mickey).  
liebt(mickey,pluto).  
liebt(minnie,mickey).  
liebt(mickey,minnie).  
  
?- liebt(X,Y).
```

```
maus(X).  
liebt(pluto,mickey).  
liebt(minnie,mickey).  
liebt(X,Y) :- liebt(Y,X).  
  
?- liebt(mickey,Y).
```

- **X** und **Y** sind Variablen. Sie können sowohl in Anfragen als auch in der Wissensbasis verwendet werden.
- Erhält der Prolog-Interpreter eine Anfrage wie `?- liebt(X,Y).` versucht er die Variablen **X** und **Y** so zu **instantiiieren** oder zu **binden** (sprich mit einem Wert zu belegen), dass die Aussage wahr wird.
- Gelingt dies, antwortet der Interpreter **true**. und gibt die gewählte erfolgreiche Instanziierung aus.
- Durch die Eingabe des Semikolons fordert man den Interpreter auf, weitere Antworten auf die Anfrage zu suchen.
- Die Klausel `maus(X).` ist ein **universeller Fakt** (Fakt mit offener Variable).



Übung

Gegeben ist die folgende Wissensbasis:

```
wizard(ron).  
hasWand(harry).  
quidditchPlayer(harry).  
wizard(X) :- hasBroom(X), hasWand(X).  
hasBroom(X) :- quidditchPlayer(X).
```

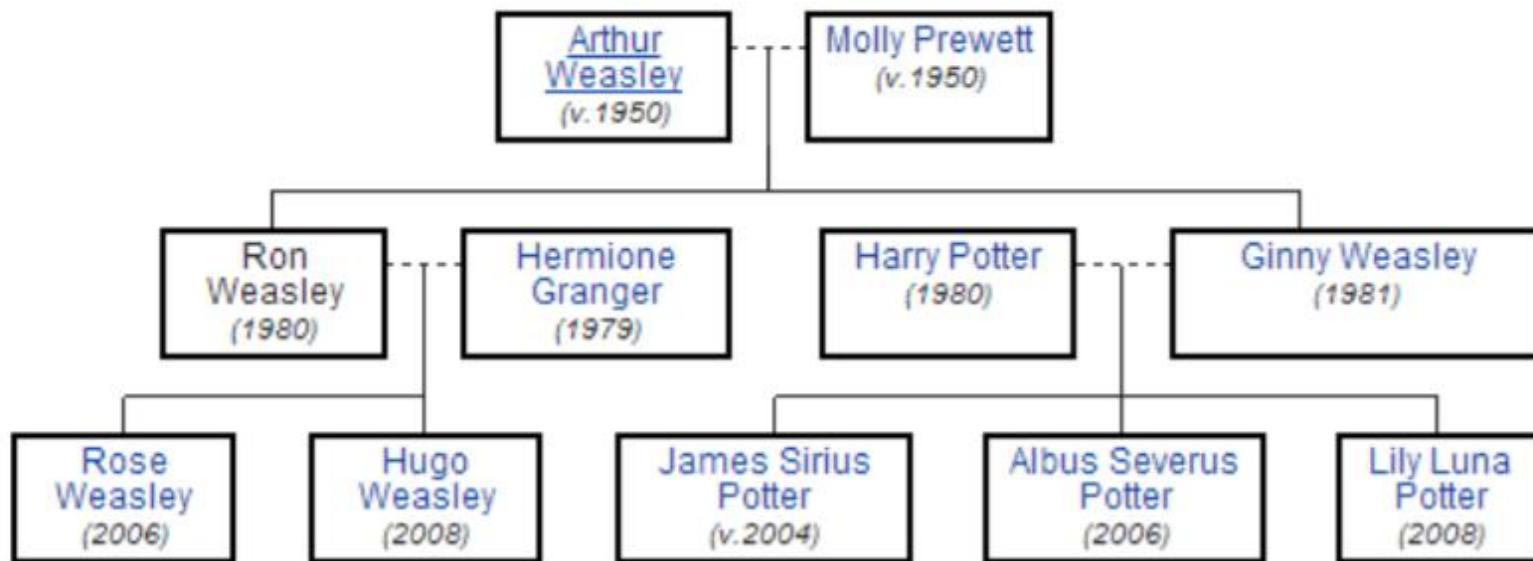
Welche Antworten gibt Prolog auf die folgenden Anfragen?

- 1 `wizard(ron)`.
- 2 `witch(ron)`.
- 3 `wizard(hermione)`.
- 4 `witch(hermione)`.
- 5 `wizard(harry)`.
- 6 `wizard(Y)`.
- 7 `witch(Y)`.



Übung Familienstammbaum

Gegeben ist folgender Familienstammbaum:



Übung Familienstammbaum

Familienstammbaum dargestellt als Prologfakten:

```
% fem/1
% fem(X): X ist feminin
fem(hermione).
fem(ginny).
fem(molly).
fem(rose).
fem(lilly_luna).

% masc/1
% masc(X): X ist maskulin
masc(arthur).
masc(ron).
masc(hugo).
masc(james_sirius).
masc(harry).
masc(albus_severus).
```

```
% et/2
% et(X,Y): X ist ein
% Elternteil von Y
et(arthur,ron).
et(arthur,ginny).
et(molly,ron).
et(molly,ginny).
et(ron,rose).
et(ron,hugo).
et(hermione,rose).
et(hermione,hugo).
et(harry,james_sirius).
et(harry,albus_severus).
et(harry,lilly_luna).
et(ginny,james_sirius).
et(ginny,albus_severus).
et(ginny,lilly_luna).
```



Übung Familienstammbaum

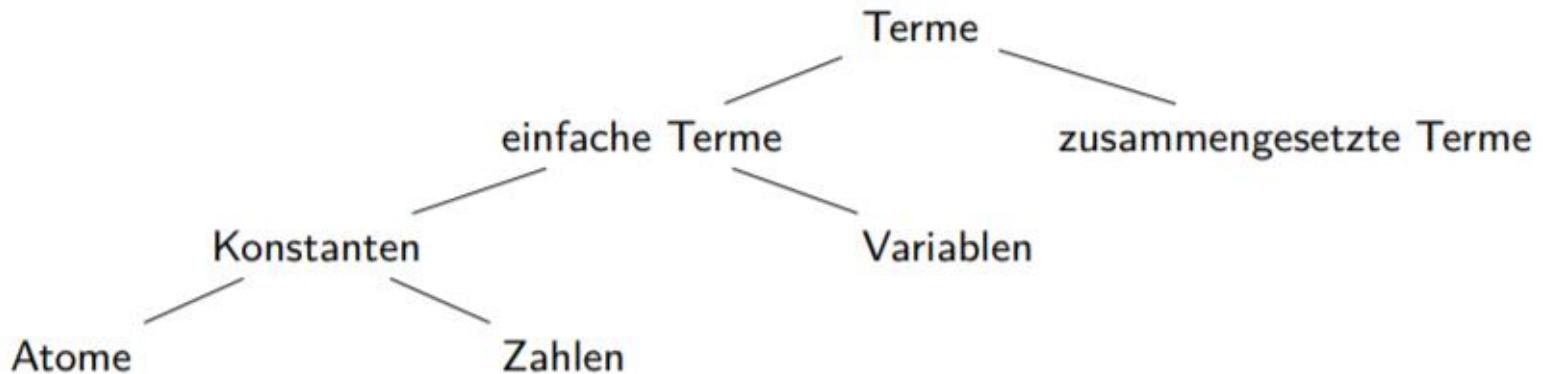
Erweitere die Wissensbasis zum Familienstammbaum um folgende
Prädikate:

```
mutter/2
vater/2
tochter/2
sohn/2
schwester/2
bruder/2
grossvater/2
```



Grundlagen

- Die grundlegende Datenstruktur in Prolog sind **Terme** (*terms*).
- Sie sind entweder **einfach** oder **zusammengesetzt**.
- Einfachen Terme in Prolog sind **Konstanten** (*constants*) und **Variablen** (*variables*)
- Die Konstanten sind **Atome** (*atoms*) und **Zahlen** (*numbers*).
- Zusammengesetzte Terme werden auch **komplexe Terme** oder **Strukturen** genannt.



Einfache Terme: Atome, Zahlen und Variablen

Atome sind Zeichenfolgen, die mit Kleinbuchstaben beginnen und nur aus Buchstaben, Zahlen und dem Unterstrich bestehen, oder Zeichenfolgen, die in Anführungszeichen stehen:
`popeye, hund13XYZ, my_dog, "Lea?!@", 'Homer Simpson'`.

Zahlen sind ganze Zahlen (*integers*) oder KommaZahlen (*floats*):
123, 89.5, 0, -323.

Variablen Variablen sind Zeichenfolgen, die mit einem Grossbuchstaben oder einem Unterstrich beginnen und nur aus Buchstaben, Zahlen und dem Unterstrich bestehen:
`X, Variable, _123, Hund_123`.

Hinweise:

- Verwenden Sie bitte immer 'sprechende' Namen für ihre Terme.
- Die Variable `_`, die nur aus dem Unterstrich besteht, ist die anonyme Variable (kommt später im Kurs). Sie sollte zunächst nicht von Ihnen verwendet werden.



Zusammengesetzte bzw. komplexe Terme

- Zusammengesetzte bzw. komplexe Terme bestehen aus einem Funktor (functor) und beliebig vielen Argumenten (arguments).
- Der Funktor ist immer ein Atom.
- Die Argumente sind einfache oder komplexe Terme.
- Beispiel für komplexe Terme: **liebt(popeye,spinat)**
- Beispiel für komplexe verschachtelte Terme:
befreundet(X,vater(vater(popeye)))
- Unter der **Stelligkeit (arity)** eines komplexen Terms versteht man die Anzahl der Argumente.



Prädikate

- Betrachtet man die Fakten einer Wissensbasis als leere Regeln, dann definieren alle Klauseln, deren Regelköpfe denselben Funktor **und** dieselbe Stelligkeit haben, zusammen ein **Prädikat**.
- **Vorsicht:** Enthält eine Wissensbasis die beiden Fakten **befreundet(popeye,pluto,garfield)** und **befreundet(X,mickey)** so definiert sie zwei verschiedene Prädikate, nämlich **befreundet/3** (3-stellig) und **befreundet/2** (2-stellig).
- Für einen guten Programmierstil beachte folgende Regeln:
 - Alle Klauseln, die zu einem Prädikat gehören, stehen direkt beieinander.
 - Die Verwendung zweier Prädikate mit identischem Funktor aber unterschiedlicher Stelligkeit geschieht nur im wohldurchdachten Ausnahmefall.
 - Jedes Prädikat wird kommentiert (Kommentarzeichen '%').

```
% liebt/2
% liebt(Person1,Person2)
liebt(pluto,mickey).
liebt(mickey,minnie).
liebt(X,Y):- liebt(Y,X).
```



- Alle Klauseln werden mit einem Punkt abgeschlossen.
- Prädikate:
 - Alle Klauseln, die zu einem Prädikat gehören, stehen direkt beieinander.
 - Die Verwendung zweier Prädikate mit identischem Funktor aber unterschiedlicher Stelligkeit geschieht nur im wohldurchdachten Ausnahmefall.
 - Jedes Prädikat wird kommentiert (Kommentarzeichen '%').
 - Die Reihenfolge der Argumente ist bedeutsam ‘kind(otto,piet)’ ist verschieden von ‘kind(piет,otto)’.
 - Wir können selber festlegen, welche Argumentposition wofür stehen sollen. Einmal getroffene Konvention beibehalten und kommentieren.



Programmieren in Prolog = Problemlösen

- Prolog ist eine deklarative Programmiersprache.
- Problemlösen in Prolog heißt Probleme beschreiben, indem Objekte mit ihren Eigenschaften und Beziehungen dargestellt werden.
 - Was ist das Problem?
 - Welche Objekte sind beteiligt?
 - Welche Eigenschaften und Beziehungen?



Zusammenfassung

Wir haben die Grundlagen und Anwendungsgebiete von Prolog kennengelernt.

- **Keywords:** Programmierparadigma, deklaratives Programmieren, Wissensbasis (Klauseln), Klauseln (Fakten, Regeln, Anfragen), Regeln (Regelkopf, Regelkörper), Prädikate, Terme (einfach und zusammengesetzt), Atome, Zahlen, Variablen, komplexe Terme (Funktor, Argument, Stelligkeit), Konjunktion, Disjunktion.
- **Wichtig:** Das deklarative Programmierparadigma von Prolog muss man verstehen.
Prolog lernt man wie alle Programmiersprachen nur durch Programmieren!



Wie löst Prolog Anfragen?

- Prolog versucht Anfragen mittels Klauseln (Fakten und Regeln) einer Wissensbasis logisch abzuleiten bzw. zu beweisen.
- Dabei wird geprüft ob das Ziel einer Anfrage (die Zielklausel) eine logische Konsequenz der Programmklaseln (Wissensbasis) ist bzw. ob sich eine Anfrage des Benutzers gegeben als Zielklausel auf Grundlage der Programmklaseln beweisen lässt.
- Das Prinzip wird **automatische Beweisführung (proof search)** genannt und beschreibt den Ansatz den Prolog bei der Suche bzw. Beantwortung von Anfragen verfolgt.



Beweisführung

- Um eine Zielklausel zu beweisen, versucht Prolog die Klausel mit den in der Wissensbasis gegebenen Fakten und Regelköpfen zu **matchen** oder zu **unifizieren**.
- Wenn die Anfrage Variablen enthält muss eine gültige Variablenbelegung (matching) gefunden werden.
- Das Prinzip der automatischen Beweisführung von Prolog basiert auf dem Prinzip der **Unifikation (unification)** und des **automatischen Rücksetzens (backtracking)**.



Matching/Unifikation

- Beim Matching handelt es sich um eine Operation, die zwei Terme miteinander vergleicht bzw. prüft ob diese durch eine geeignete Variablenbelegung gleichgesetzt (unifiziert) werden können.
- Das Matching ist ein Teil der automatischen Beweisführung. Es gibt jedoch auch das eingebaute Prädikat (auch **Unifikationsoperator** genannt) `=`, welches zwei Terme matcht.

Matchingregel

Zwei Terme matchen genau dann, wenn sie gleich sind oder wenn sie Variablen beinhalten, die so belegt werden können, dass die beiden Terme gleich werden.



Matching einfacher Terme: Konstanten

Zwei Konstanten matchen genau dann, wenn sie gleich sind.

```
?- =(popeye,popeye).  
true.
```

```
?- =(popeye,'Popeye').  
false.
```

```
?- =(popeye,'popeye').  
true.
```

```
?- =(12,12).  
true.
```

```
?- =(12,'12').
```

Das Prädikat = kann auch in der Infixnotation genutzt werden:

```
?- regen = schnee.  
false.
```



Matching einfacher Terme: Variablen

- Wenn einer der Terme eine Variable ist, dann kann die Variable mit dem anderen Term belegt werden und beide Terme matchen.
- Dies funktioniert unabhängig davon, ob der andere Term einfach oder komplex ist.
- Besteht die Anfrage aus mehr als einer elementaren Zielklausel, muss zusätzlich die Variablenbelegung aller Elementarklauseln kompatibel sein.

```
?- = (mag_spinat(popeye), X).
```

```
X=mag_spinat(popeye).
```

```
?- X=Y, X=popeye.
```

```
X = popeye,
```

```
Y = popeye.
```

```
?- X=popeye, X=pluto.
```

```
false.
```



Matching komplexer Terme

Komplexe Terme matchen genau dann wenn:

- ① die Terme den gleichen Funktor und dieselbe Stelligkeit haben **und**
- ② alle korrespondierenden Argumente matchen **und**
- ③ die Variablenbelegungen miteinander kompatibel sind.

```
?- kill(shoot(gun),Y) = kill(X,stab(knife)).  
X = shoot(gun),  
Y = stab(knife).  
  
?- kill(shoot(gun), stab(knife)) = kill(X,stab(Y)).  
X = shoot(gun),  
Y = knife.  
  
?- mag(X,X) = mag(popeye,pluto).  
false.
```



Übung: Matching

Welche der folgenden Paare von Termen matchen?

- 1 ?- **bread** = **bread**.
- 2 ?- 'Bread' = **bread**.
- 3 ?- 'bread' = **bread**.
- 4 ?- **Bread** = **bread**.
- 5 ?- **bread** = **sausage**.
- 6 ?- **food(bread)** = **bread**.
- 7 ?- **food(bread)** = **X**.
- 8 ?- **food(X)** = **food(bread)**.
- 9 ?- **food(bread,X)** = **food(Y,sausage)**.
- 10 ?- **food(bread,X,beer)** = **food(Y,sausage,X)**.
- 11 ?- **food(bread,X,beer)** = **food(Y,kahuna_burger)**.
- 12 ?- **food(X)** = **X**.
- 13 ?- **meal(food(bread),drink(beer))** = **meal(X,Y)**.
- 14 ?- **meal(food(bread),X)** = **meal(X,drink(beer))**.



Matching: Definition

Seien $term1$ und $term2$ zwei Terme.

- ① Wenn $term1$ und $term2$ Konstanten sind, matchen sie genau dann, wenn sie das gleiche Atom oder die gleiche Zahl sind.
- ② Wenn $term1$ eine Variable ist, dann matchen $term1$ und $term2$. Die Variable $term1$ wird dann mit dem Term ($term2$) instantiiert (analog für den Fall, dass $term2$ eine Variable ist).
- ③ Wenn $term1$ und $term2$ komplexe Terme sind, dann matchen sie genau dann, wenn:
 - die Terme den gleichen Funktor und dieselbe Stelligkeit haben **und**
 - alle korrespondierenden Argumente matchen **und**
 - die Variablenbelegungen miteinander kompatibel sind.

Wenn keine der drei Gegebenheiten zutrifft, matchen die beiden Terme nicht.



Unifikationsoperator

Neben dem Unifikationsoperator

```
sonne = regen.  
=(sonne,regen).
```

gibt es auch den negierten Unifikationsoperator

```
sonne \= regen.  
\=(sonne,regen).
```

Der Operator $\backslash=/2$ gelingt genau dann, wenn der Unifikationsoperator $=/2$ scheitert.



Übung: Die Negation des Unifikationsoperators

Welche der folgenden Anfragen führen zu **true**.?

- 1 ?- **a** \= **a**.
- 2 ?- '**a**' \= **a**.
- 3 ?- **A** \= **a**.
- 4 ?- **f(a)** \= **a**.
- 5 ?- **f(a)** \= **A**.
- 6 ?- **f(A)** \= **f(a)**.
- 7 ?- **g(a,B,c)** \= **g(A,b,C)**.
- 8 ?- **g(a,b,c)** \= **g(A,C)**.
- 9 ?- **f(X)** \= **X**.



Prologmatching vs. Standardunifikation

zyklische Anfrage:

```
?- vater(X) = X.
```

In der Auswertung dieser Anfrage unterscheidet sich das Matching von Prolog von der Standardunifikation:

Antwort der Standardunifikation

Die Terme matchen **nicht**. Egal mit was man die Variable X belegt (z.B. $X=vater(vater(lena))$), hat der linke Term immer eine Klammerungsebene mehr als der rechte Term.

Standardunifikationsalgorithmen sind pessimistisch und prüfen vor jeder Unifikation, ob die zu unifizierenden Terme einen Zyklus aufbauen (occurs check).



Prologmatching vs. Standardunifikation

zyklische Anfrage:

```
?- vater(X) = X.
```

In der Auswertung dieser Anfrage unterscheidet sich das Matching von Prolog von der Standardunifikation:

Antwort von Prolog

Die Antwort hängt ab von der Prologimplementierung. Ältere Implementierungen erzeugen Auskünfte wie

Not enough memory to complete query!.

Neuere Implementierungen wie SWI-Prolog matchen die beiden Terme:

```
?- vater(X) = X.  
X=vater(X)
```

Da Matching so zentral für Prolog ist und so häufig passiert ist Prolog optimistisch und erwartet, dass es nicht mit gefährlichen, sprich zyklischen Strukturen gefüttert wird.



Internrepräsentation zyklischer Strukturen

Betrachte den trace zu den folgenden Anfragen:

```
?- X = vater(X), Y=X.  
?- X = mag(X,Y), Y=X.
```

Der Tracemode wird mit **trace**. ein- und mit **notrace**. ausgeschaltet.

Vermeiden Sie bei der Proglogprogrammierung solche zyklischen
Definitionen!



Programmieren mit Matching

```
vertical(line(point(X,Y),point(X,Z))).  
horizontal(line(point(X,Y),point(Z,Y))).
```

Was definiert diese Wissensbasis?

Einige Anfragen:

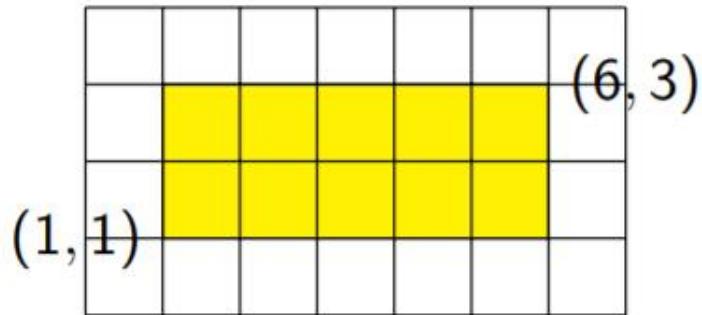
```
?- vertical(line(point(1,3),point(1,8))).  
?- horizontal(line(point(1,3),point(4,Y))).  
?- horizontal(line(point(1,3),point(Z,3))).  
?- horizontal(line(point(1,3),point(Z,4))).  
?- horizontal(line(point(1,3),P)).
```



Übung: Programmieren mit Matching

```
vertical(line(point(X,Y),point(X,Z))).  
horizontal(line(point(X,Y),point(Z,Y))).
```

Können Sie zu dieser Wissensbasis eine Anfrage formulieren, mit der sie die fehlenden Eckkoordinaten dieses Rechtecks berechnen können?



Regeln in Prolog

Regelform in Prolog:

`ich_muede :- abend, dunkel.`

Konklusion ← Prämisse UND Prämisse

`abend, dunkel → ich_muede`

Wenn es Abend ist und dunkel ist, dann bin ich müde.

Modus Ponens:

`abend.`

`dunkel.`

`ich_muede :- abend, dunkel.`

`ich_muede.`



Regeln und die Beweissuche

- Angenommen die Wissensbasis enthält eine Regel $B :- A$ (d.h. wenn A gilt, dann gilt auch B) und es soll B bewiesen werden.
- Beweissuche:
 - ➊ B soll bewiesen werden (es erfolgt eine Anfrage zu B).
 - ➋ Matcht ein Fakt oder der Kopf einer Regel aus der Wissensbasis mit B ?
 - ➌ Ja und die Regel besagt das B wahr ist, wenn A auch wahr ist.
 - ➍ Kann A bewiesen werden?
 - ➎ Wenn A mit einem Fakt aus der Wissensbasis matcht (d.h. A ist wahr), so folgt, dass auch die Anfrage nach B wahr ist.



Beweisstrategie des Interpreters

- Anfrage gilt als zu beweisende **Behauptung**.
- **Head-Matching**: Anfrage **unifiziert** mit Kopf einer Klausel (also mit linker Regelseite oder Fakt).
- Unifikation liefert **Variablenbelegungen**.
- Klauselrumpf wird bewiesen.
- Alternative Lösungen über **Backtracking**.
- Reihenfolge der **Suchraumtraversierung**:
top-down depth-first left-to-right



Headmatching

Ein Prädikat aus einer Anfrage muß mit dem Kopf einer Klausel unifizierbar sein.

Anfrage: **?- sterblich (sokrates).**

Klausel: **sterblich (X) :- menschlich (X).**



Headmatching

Ein Prädikat aus einer Anfrage muß mit dem Kopf einer Klausel unifizierbar sein.

Anfrage: **?- sterblich (sokrates) .**



Klausel: **sterblich (X) :- menschlich (X) .**



Headmatching

Ein Prädikat aus einer Anfrage muß mit dem Kopf einer Klausel unifizierbar sein.

Anfrage: **?- sterblich (sokrates).**



Klausel: **sterblich (X) :- menschlich (X).**

Variableninstanziierung: **X=sokrates**

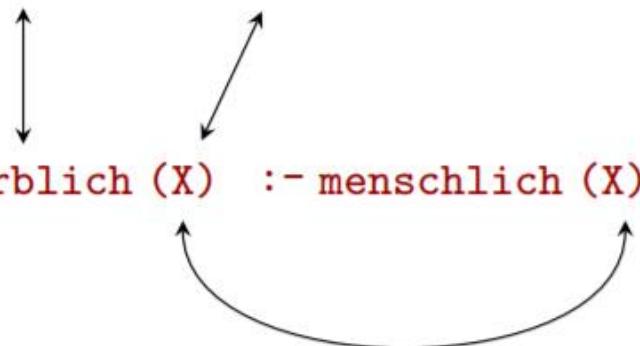


Headmatching

Ein Prädikat aus einer Anfrage muß mit dem Kopf einer Klausel unifizierbar sein.

Anfrage: **?- sterblich (sokrates).**

Klausel: **sterblich (X) :- menschlich (X).**



Variableninstanziierung: **X=sokrates**

Neue Anfrage: **?- menschlich(sokrates).**



Top-down Verfahren

- Das Head-Matching wird im top-down- Verfahren durchgeführt.
- Der Interpreter durchsucht die Datenbasis von oben nach unten, um passende Klauseln für einen Beweis zu finden.

```
mag_spinat(popeye).  
hat_trainiert(garfield).  
ist_stark(X) :- hat_trainiert(X).  
ist_stark(X) :- mag_spinat(X).
```

Was ist die erste Antwort auf die Frage:

```
?- ist_stark(X).
```



Innerhalb von Regeln: left to right

- Regelrümpfe werden von links nach rechts bewiesen (left-to-right).
- Erst wenn ein Beweis für das i-te Prädikat im Rumpf gefunden ist, kann das i+1-te Prädikat bewiesen werden.

Beispiel:

```
schwester(X,Y) :-  
    fem(X),  
    geschwister(X,Y).
```

Zunächst wird `fem(X)`, dann `geschwister(X,Y)` bewiesen.



Backtracking: Depth-first

Backtracking kann durch zwei Ursachen ausgelöst werden:

- Der aktuelle Beweis ist in einer **Sackgasse**.
- Eine **alternative Lösung** soll berechnet werden.

In jedem Fall geht der Interpreter zur letzten Verzweigung im Beweisbaum zurück, an der noch Alternativen offen waren (depth-first).

```
mag_spinat(popeye).  
hat_trainiert(garfield).  
ist_stark(X) :- hat_trainiert(X).  
ist_stark(X) :- mag_spinat(X).
```

Betrachte die folgenden Anfragen im Tracemode:

```
?- ist_stark(popeye).  
?- ist_stark(X).
```



Zusammenfassung

- Die Beweisführung in Prolog erfolgt durch die Strategie der Tiefensuche.
- Teilziele einer Anfrage werden von links nach rechts bearbeitet.
- Für jedes Teilziel wird die erste Klausel (von oben nach unten) ausgewählt.
- Ist die Klausel ein Fakt, versucht Prolog die Anfrageklausel mit dem Fakt zu matchen. Gelingt dies, so ist das Teilziel bewiesen.
- Ist die Klausel eine Regel, versucht Prolog das Teilziel mit dem Regelkopf zu matchen. Gelingt dies, so versucht Prolog den Regelkörper zu beweisen (der Regelkörper ersetzt das Teilziel). Gelingt dies auch, so ist die Anfrageklausel bewiesen.
- Sollte die Beweisführung aufgrund einer unmöglichen Unifikation scheitern, springt Prolog zu dem letzten Punkt zurück, an dem eine Entscheidung getroffen wurde (Backtracking).
- Beim Backtracking werden die gemachten Variablenbindungen aufgehoben und nach einer alternativen Klausel gesucht.
- Findet Prolog keinen Fakt und keine Regel mit dem die Anfrage bewiesen werden kann, so wird **false** zurückgegeben.



Beispiel einer Beweissuche

- Beweissuche für die Anfrage: `hund(pluto)`.

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```



Suchbaum einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```

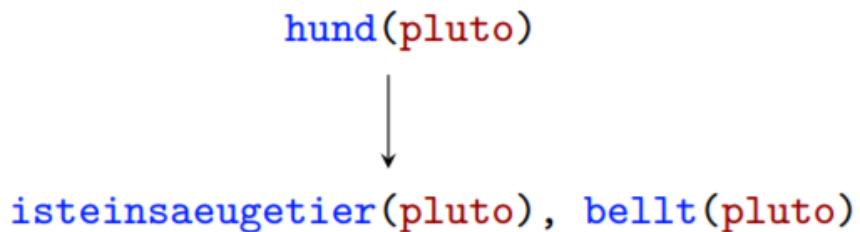
hund(pluto)

Warum sagt man das Prolog die Strategie Tiefensuche anwendet?



Suchbaum einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```

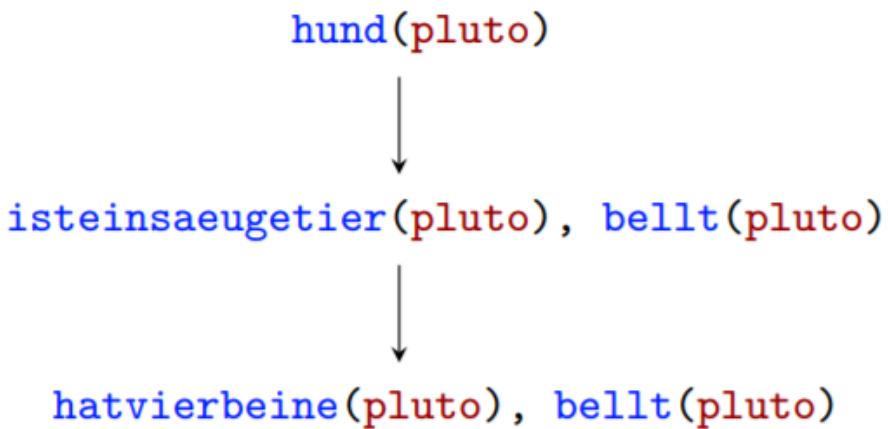


Warum sagt man das Prolog die Strategie Tiefensuche anwendet?



Suchbaum einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```

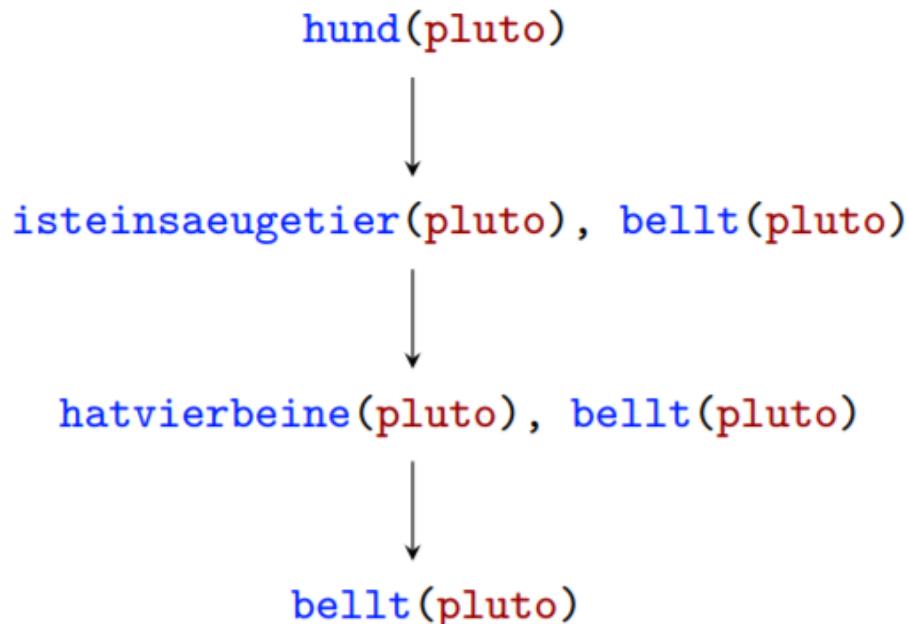


Warum sagt man das Prolog die Strategie Tiefensuche anwendet?



Suchbaum einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```

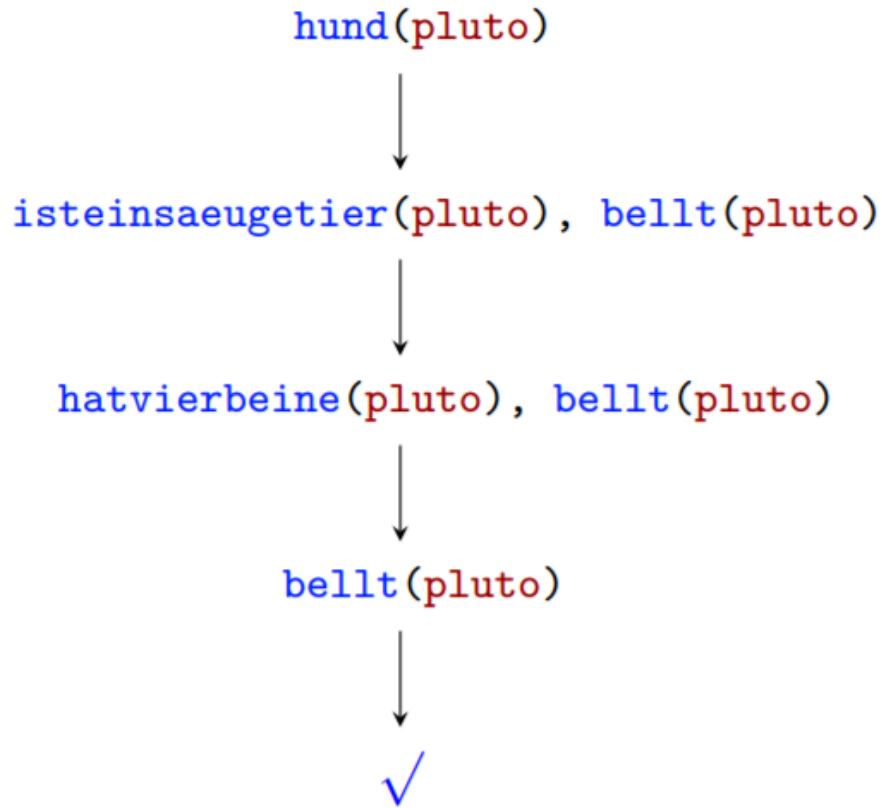


Warum sagt man das Prolog die Strategie Tiefensuche anwendet?



Suchbaum einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```



Warum sagt man das Prolog die Strategie Tiefensuche anwendet?



Trace einer Beweissuche

```
istehrlich(pluto).  
hatvierbeine(pluto).  
bellt(pluto).  
  
hund(X) :-  
    isteinsaeugetier(X),  
    bellt(X).  
  
isteinsaeugetier(X) :-  
    hatvierbeine(X).  
  
?- hund(pluto).
```

```
[trace] 8 ?- hund(pluto).  
Call: (7) hund(pluto) ?  
Call: (8) isteinsaeugetier(pluto) ?  
Call: (9) hatvierbeine(pluto) ?  
Exit: (9) hatvierbeine(pluto) ?  
Exit: (8) isteinsaeugetier(pluto) ?  
Call: (8) bellt(pluto) ?  
Exit: (8) bellt(pluto) ?  
Exit: (7) hund(pluto) ?  
  
true.
```



Beispiel einer Beweissuche: Entscheidungspunkt

- Beweissuche für die Anfrage: `term(X)`.

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :- term1(X), term2(X), term3(X).  
  
?- term(X).
```



Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



Beispiel einer Beweissuche: Entscheidungspunkt

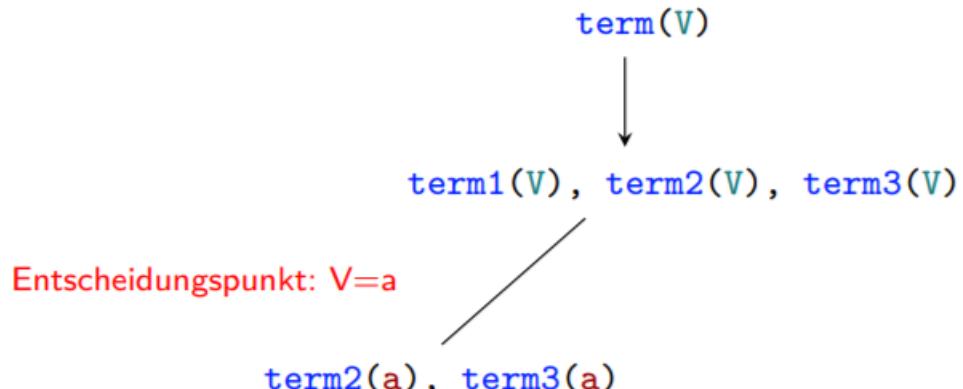
```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```

term(V)
↓
term1(V), term2(V), term3(V)



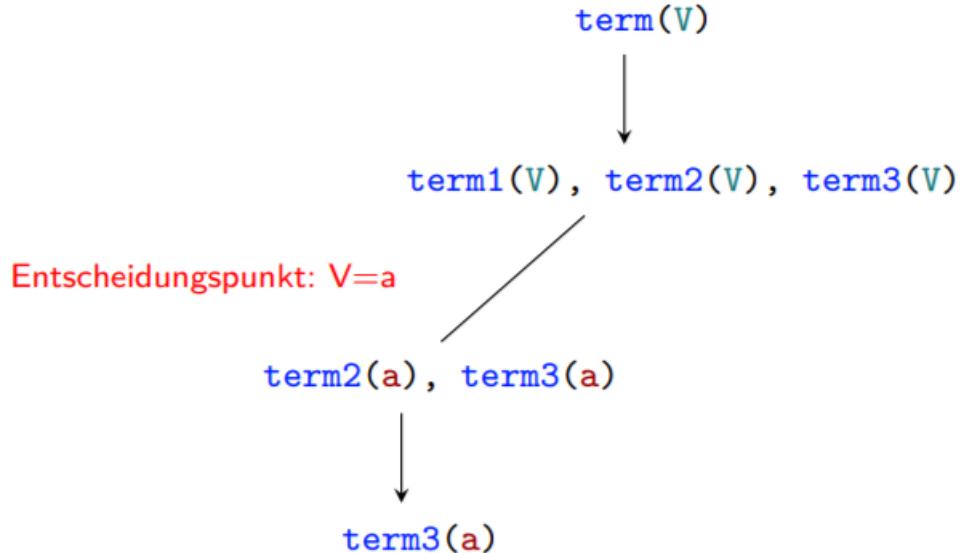
Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



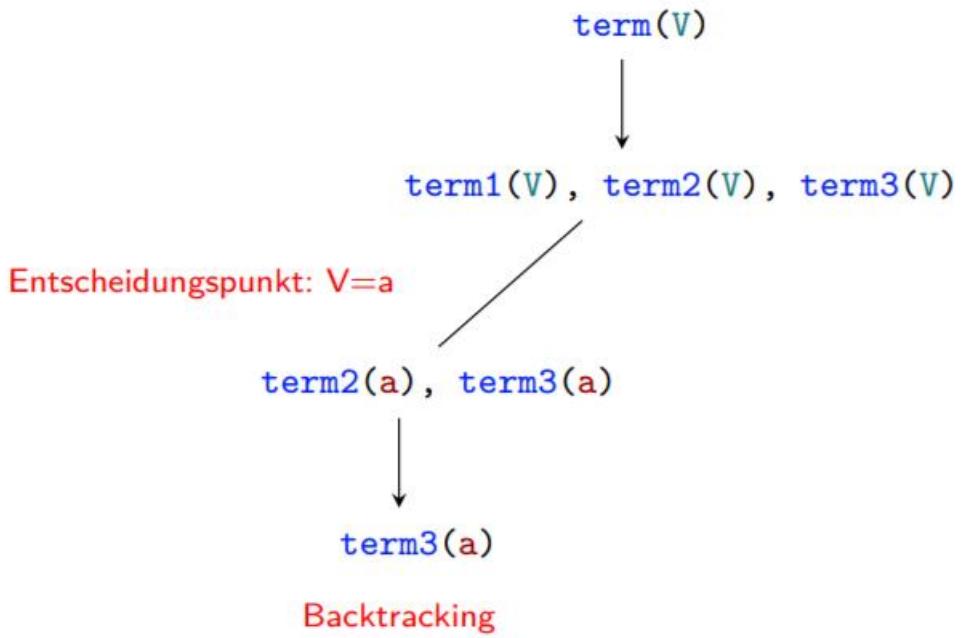
Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



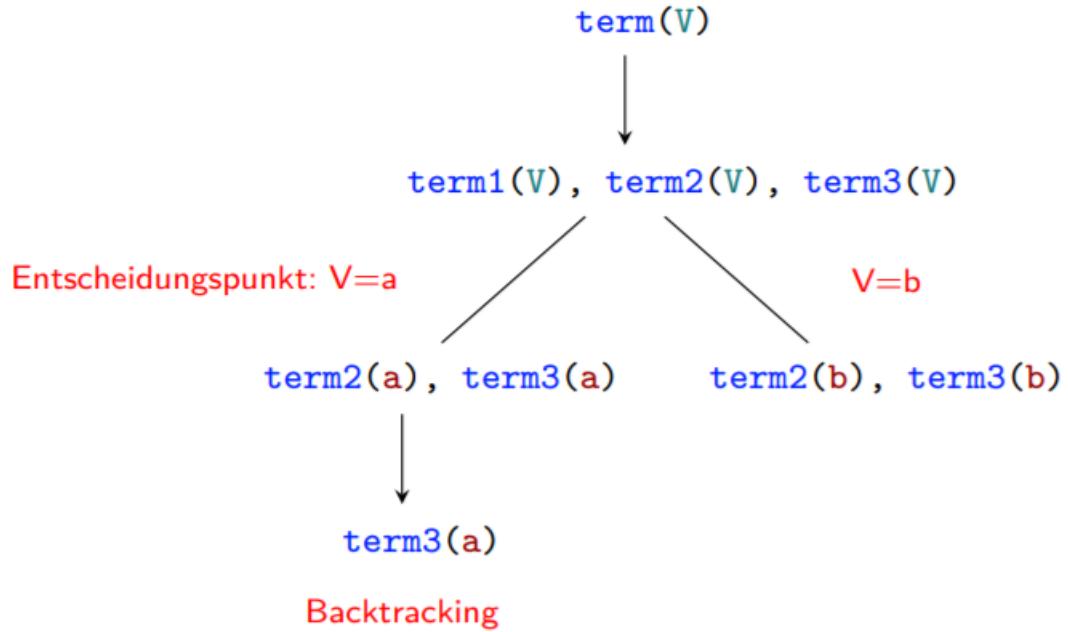
Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



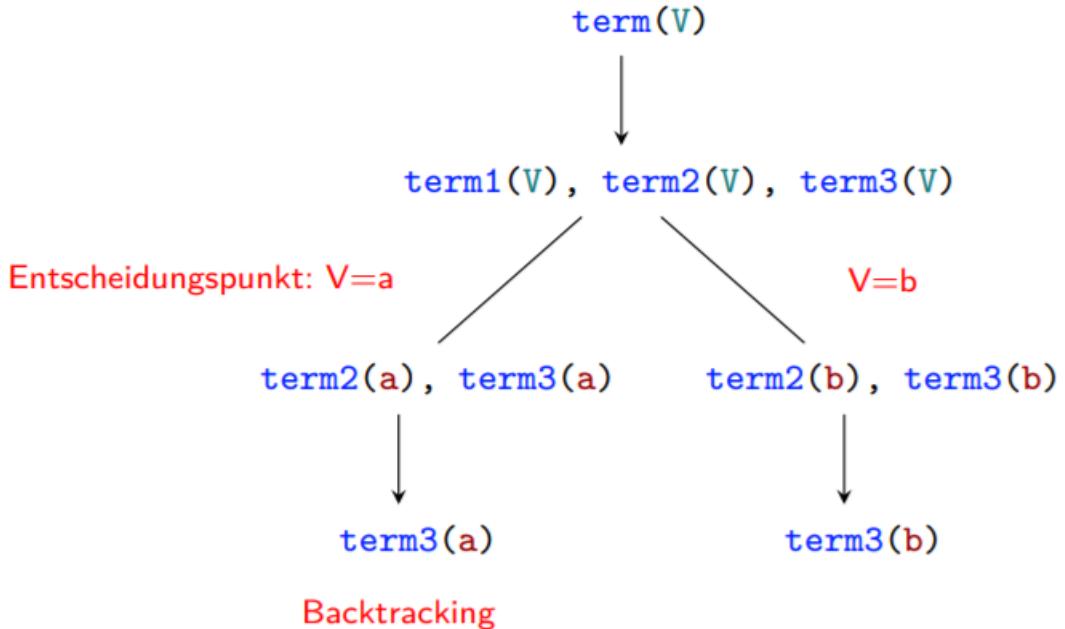
Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



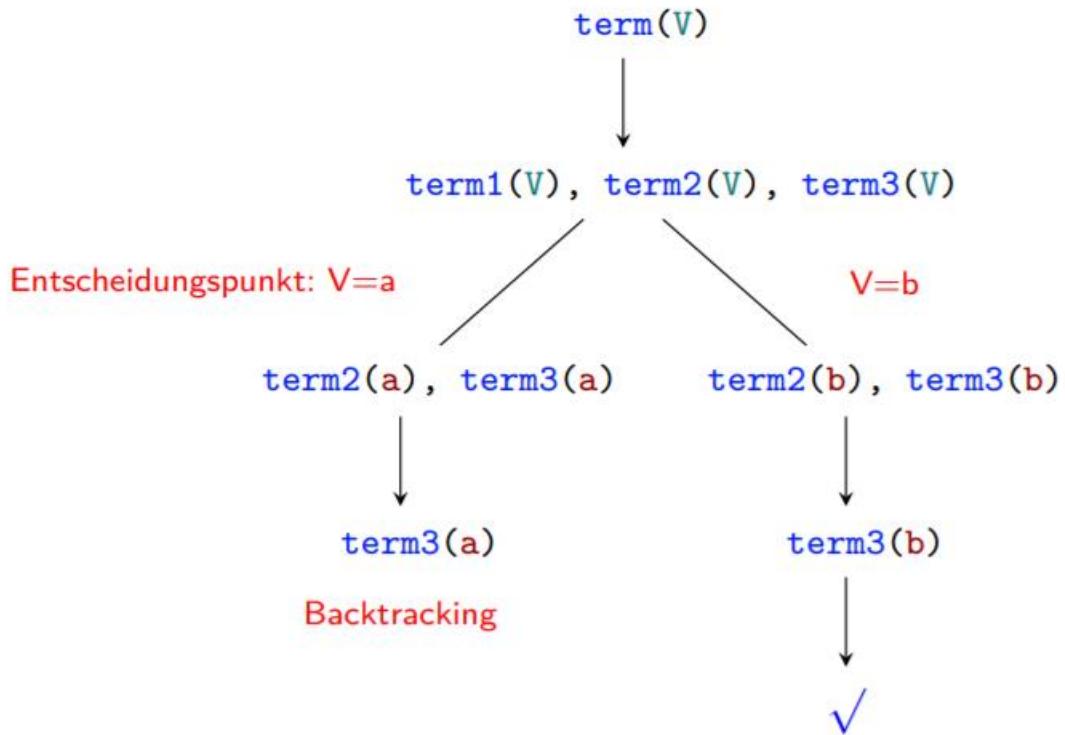
Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```



Beispiel einer Beweissuche: Entscheidungspunkt

```
term1(a).  
term1(b).  
  
term2(a).  
term2(b).  
  
term3(b).  
  
term(X) :-  
    term1(X),  
    term2(X),  
    term3(X).  
  
?- term(X).
```

```
[trace] 19 ?- term(X).  
Call: (7) term(_G11414) ?  
Call: (8) term1(_G11414) ?  
Exit: (8) term1(a) ?  
Call: (8) term2(a) ?  
Exit: (8) term2(a) ?  
Call: (8) term3(a) ?  
Fail: (8) term3(a) ?  
Redo: (8) term1(_G11414) ?  
Exit: (8) term1(b) ?  
Call: (8) term2(b) ?  
Exit: (8) term2(b) ?  
Call: (8) term3(b) ?  
Exit: (8) term3(b) ?  
Exit: (7) term(b) ?  
  
X = b.
```



Zusammenfassung

- Wir haben gelernt wie komplexe Strukturen durch Matching in Prolog aufgebaut werden können und wie die Beweisführung in Prolog funktioniert.
- Keywords: Beweisführung, Beweisstrategie (top-down, left-to-right, depth-first), Matching, Unifikation, Backtracking.
- Wichtig: Der Ablauf des Matchings und der Beweisführung (inkl. Backtracking) sind essentiell für die Programmierung in Prolog.

