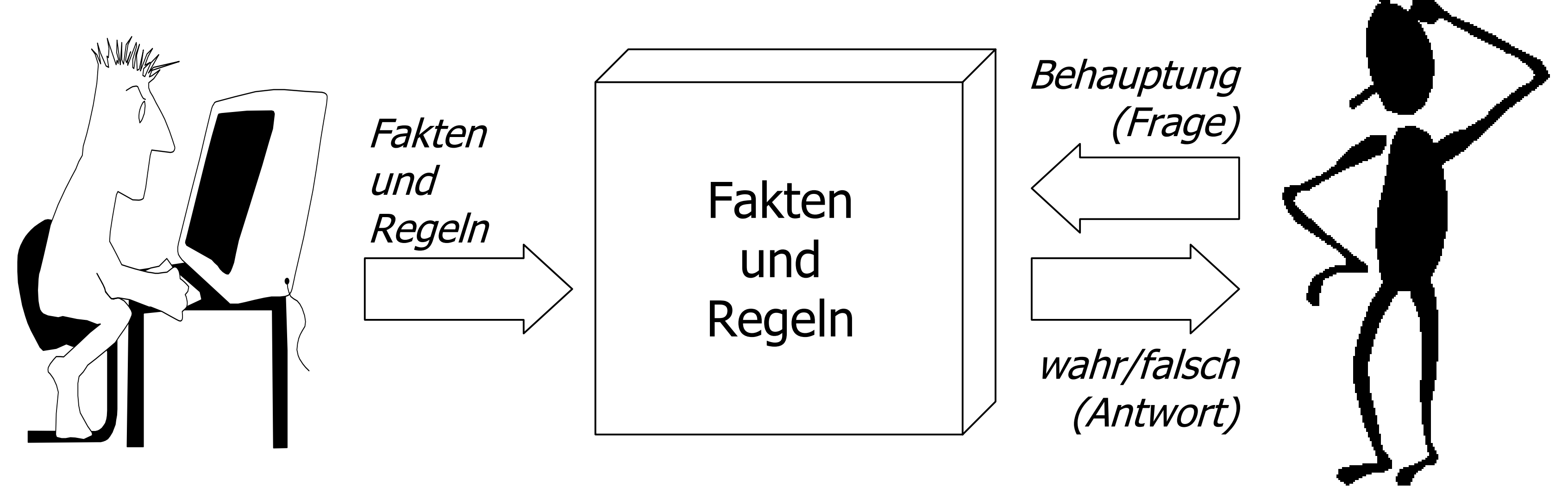
Eine Einführung in Prolog und die logische Programmierung

Mala Bachmann

Januar 2001

Programmieren in Prolog



Prolog-
Progammierer

Prolog-
Programiersystem

Benutzer

Programmieren in Prolog

Programmieren in Prolog bedeutet

- **Tatsachen (Fakten)** über Objekte und deren Beziehungen zu deklarieren

  Sokrates ist ein Mensch.

- **Regeln** über Objekte und deren Beziehungen zu definieren

  Alle Menschen sind sterblich.

- **Fragen** zu den Objekten und Beziehungen zu stellen

  Ist Sokrates sterblich?

Prozedurale versus logische (deklarative) Programmierung

Prozedural

Der Benutzer sagt, *wie* das Problem gelöst werden soll.

Beschreibung eines Kreises:

Resultat einer 360 Grad Rotation mit dem Zirkel.

Deklarativ

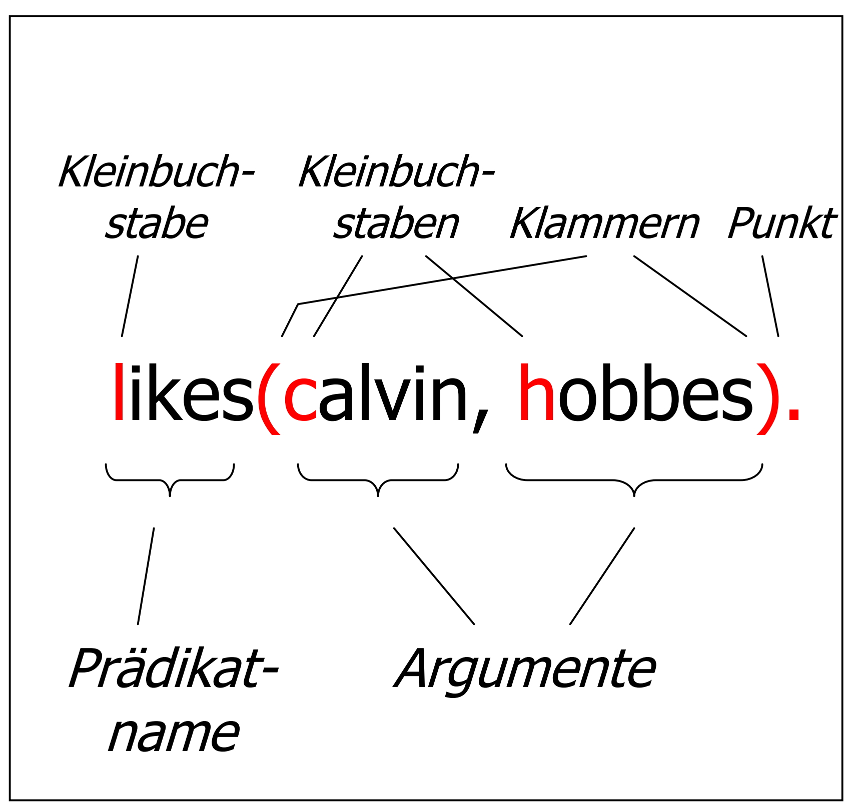
Der Benutzer beschreibt, *was* das Problem ist. *Wie* es gelöst wird, wird vom System kontrolliert.

Beschreibung eines Kreises:

Menge aller Punkte, die von einem vorgegebenen Punkt denselben Abstand hat.

Ablauf / Ziele

Prolog Essentials
Fakten



Weitere Beispiele von Fakten:

tiger(hobbes).

person(calvin).

dreams(calvin).

lazy(calvin).

gives(calvin, ball, hobbes).

...

Prolog Essentials
Fragen

?- likes(calvin, hobbes).
yes
?- tiger(hobbes).
yes
?- tiger(calvin).
no
?- lion(calvin).
no

```
tiger(hobbes).

person(calvin).
person(susie).

likes(calvin, hobbes).
likes(susie, school).
likes(susie, hobbes).
```

Prolog Essentials
Variablen

?- likes(calvin, X).
X=hobbes;
No

Grossbuch-staben

?- likes(Who, hobbes).
Who=calvin;
Who=susie;
no

```
tiger(hobbes).

person(calvin).
person(susie).

likes(calvin, hobbes).
likes(susie, school).
likes(susie, hobbes).
```

Prolog Essentials
Konjunktionen

„und"

Mögen sich Calvin und Hobbes gegenseitig?

```
?- likes(calvin, hobbes), likes(hobbes, calvin).
```

Gibt es etwas, das sowohl Calvin als auch Susie mögen?

```
?- likes(susie, X), likes(calvin, X).
```

```
tiger(hobbes).

person(calvin).
person(susie).

likes(calvin, hobbes).
likes(susie, school).
likes(susie, hobbes).
```

Prolog Essentials
Beantwortungsmechanismus (1)

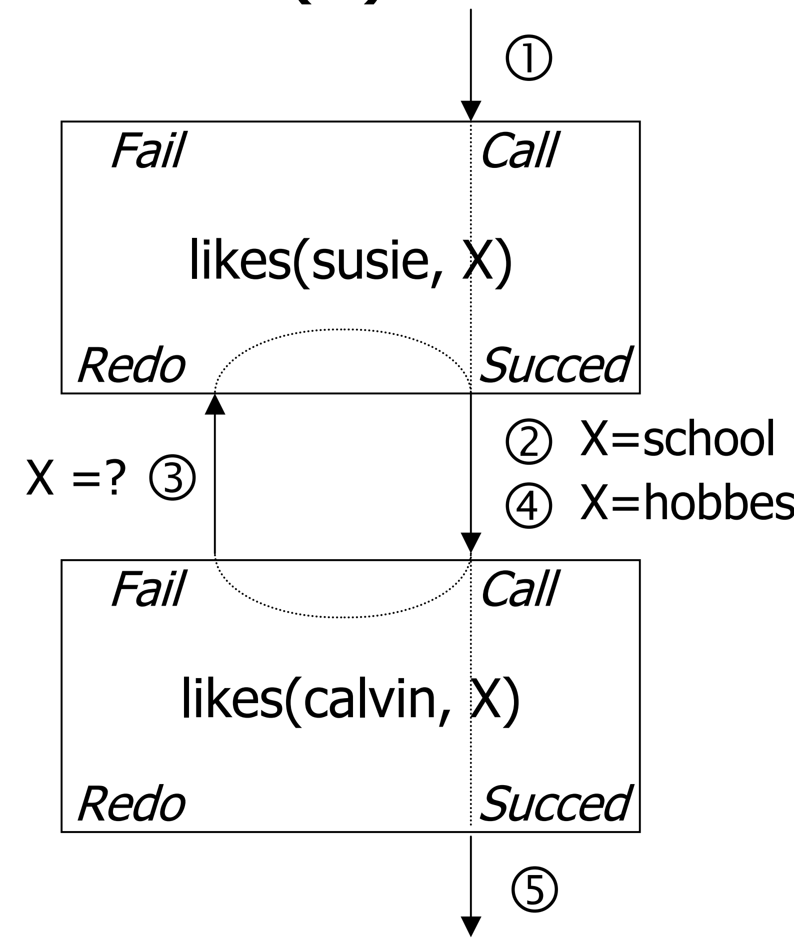
```
tiger(hobbes).

person(calvin).
person(susie).

likes(calvin, hobbes).
likes(susie, school).
likes(susie, hobbes).
```

```
?- likes(susie, X), likes(calvin, X).
```

zu beweisende Goals

X=hobbes;

no



Prolog Essentials
Regeln

„falls"

Alle Personen mögen Hobbes.

likes(X, hobbes) :- person(X).

Head *Body*

„Ein Objekt mag Hobbes, falls dieses Objekt eine Person ist."

Weitere Beispiele von Regeln:

likes(X, hobbes) :- person(X), nice(X).

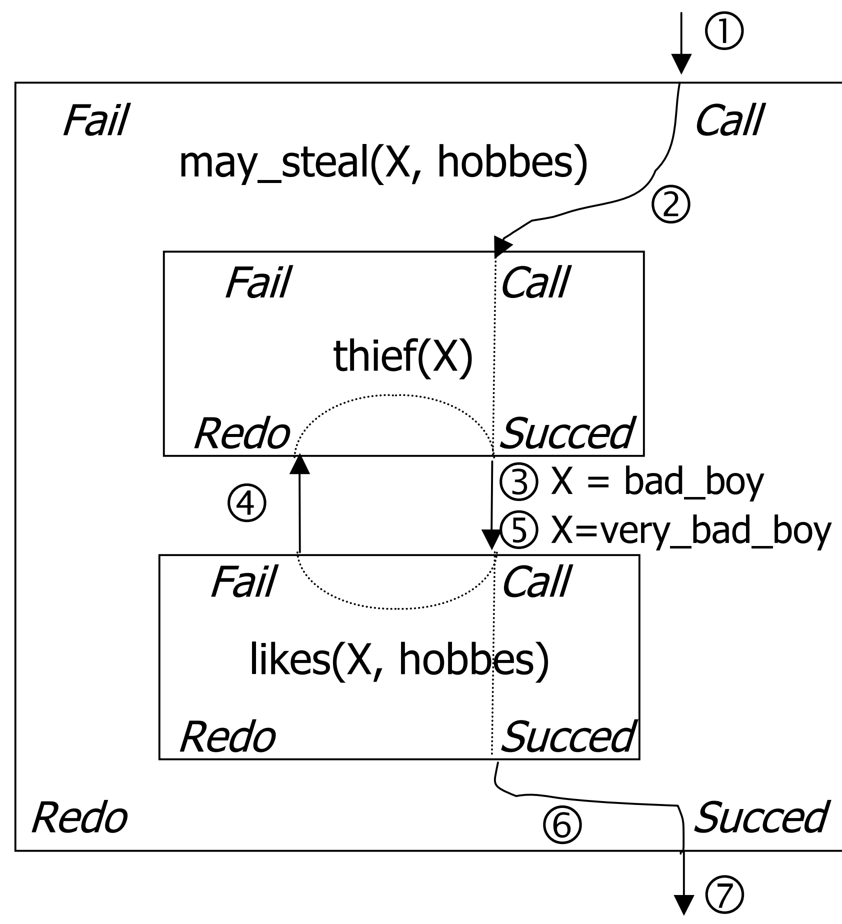
likes(X, hobbes) :- tiger(X).

may_steal(Person, Thing) :- thief(Person), likes(Person, Thing).

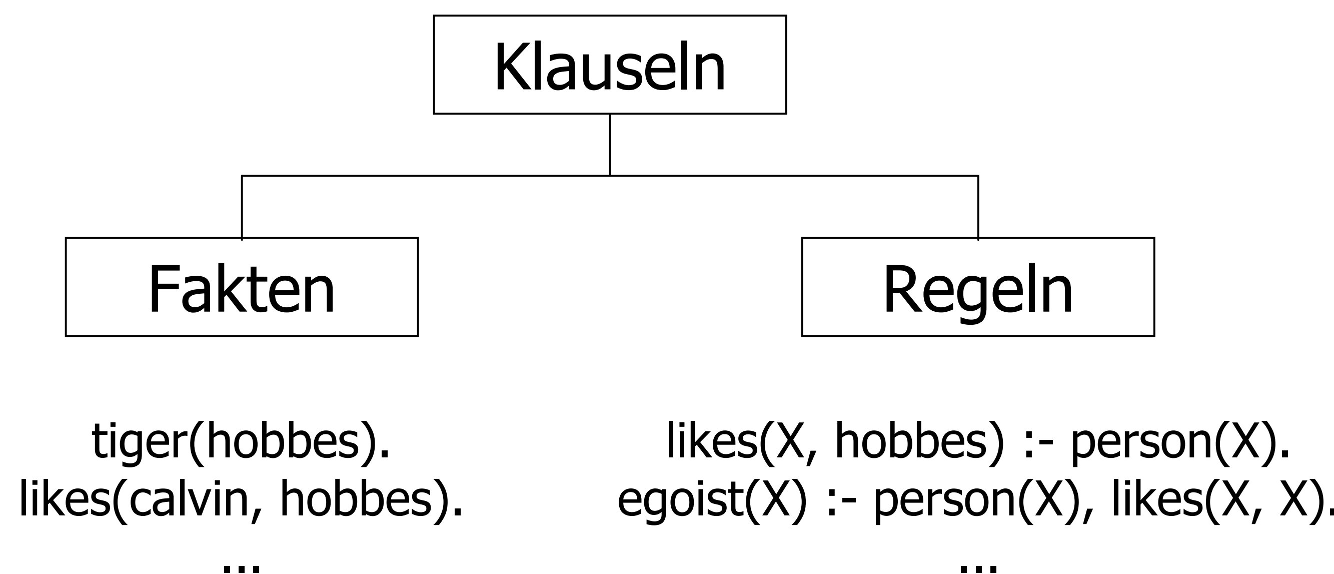
Prolog Essentials
Beantwortungsmechanismus (2)

```
thief(bad_boy).
thief(very_bad_boy).

likes(calvin, hobbes).
likes(bad_boy, money).
likes(very_bad_boy, hobbes).

may_steal(Person, Thing) :-
        thief(Person),
        likes(Person, Thing).
```

?- may_steal(X, hobbes).
X=very_bad_boy



Prolog Essentials
Klauseln



Prolog Essentials

Übung: Familien-Datenbank

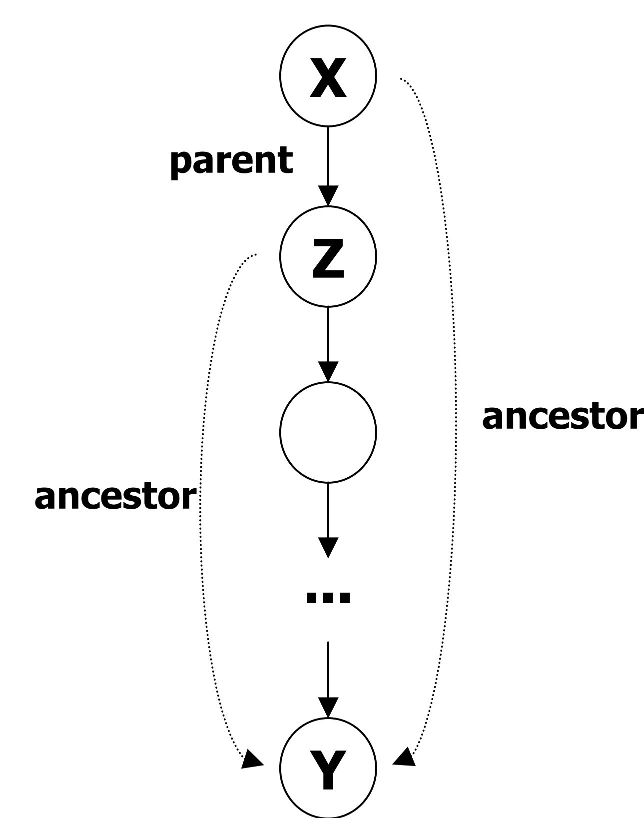
Prolog Essentials
Rekursive Regeln (1)

```
ancestor(X, Y) :-
        parent(X, Y).
ancestor(X, Y) :-
        parent(X, Z),
        ancestor(Z, Y).
```

Abbruch-bedingung (for the first clause)

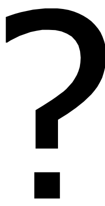
Rekursion (for ancestor(Z, Y))

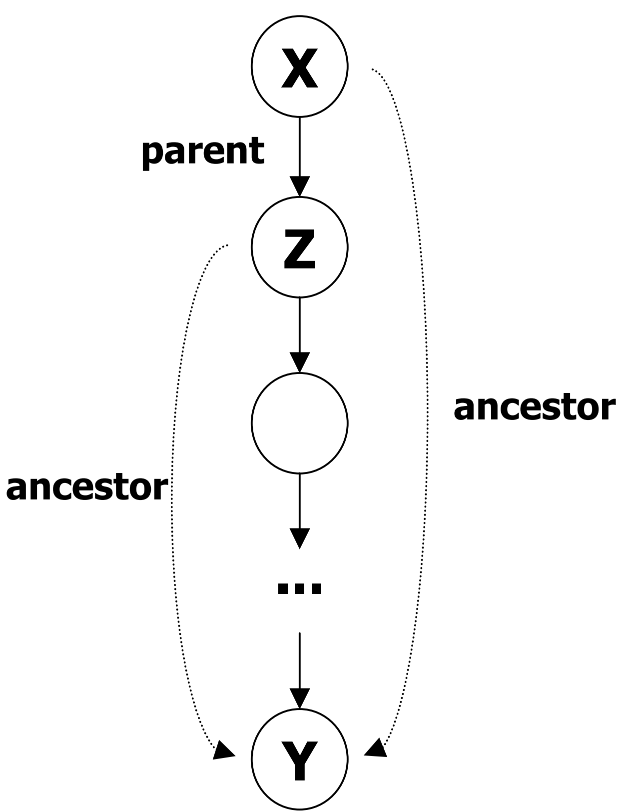
Eine Prozedur ist *rekursiv*, wenn in einer Regel das Prädikat, durch das die Prozedur definiert ist, wieder aufgerufen wird.



Prolog Essentials
Rekursive Regeln (2)

```
ancestor(X, Y) :-
        ancestor(Z, Y),
        parent(X, Z).
ancestor(X, Y) :-
        parent(X, Y).
```

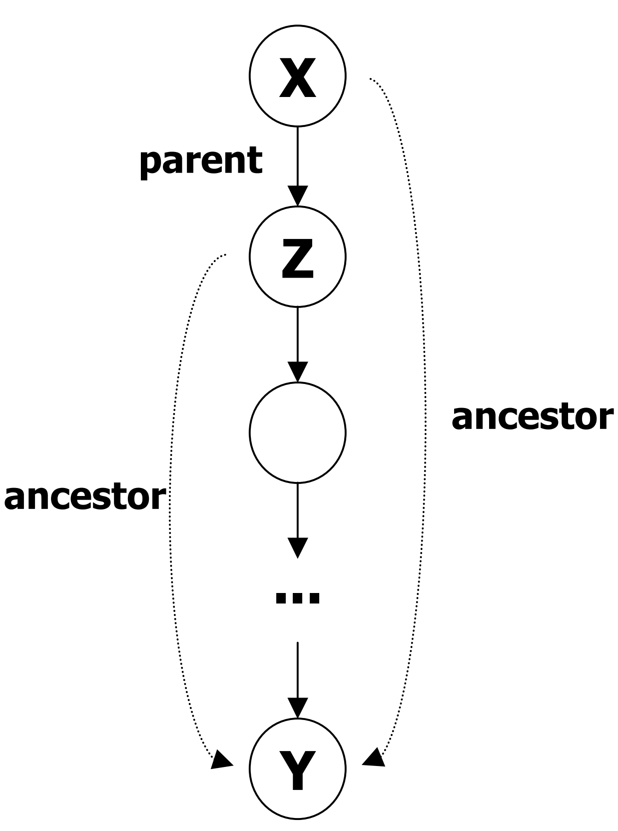




Prolog Essentials
Rekursive Regeln (3)

```
ancestor(X, Y) :-
        ancestor(Z, Y),
        parent(X, Z).
ancestor(X, Y) :-
        parent(X, Y).
```

Die Prozedur ist vom deklarativen Standpunkt aus korrekt. Sie führt aber mit der Prolog-Auswertungsstrategie zu einer Endlosschleife



Prozedurale Komponente von Prolog

H :- B1, B2.

Deklarative Semantik	Prozedurale Semantik
H erfüllt, falls sowohl B1 als auch B2 erfüllt werden können	H erfüllt, falls zuerst B1 und dann B2 erfüllt werden können

Für die Abarbeitung der Prolog-Programme wird den Programmklauseln eine prozedurale Semantik unterlegt.

Logische Programmierung

Programm	Menge von Formeln (Klauseln)
Berechnung	Beweis einer Formel (eines Ziels/Goals) mit Hilfe des Programms

Anwendungen

- Expertensysteme
- Sprachverarbeitung
- Symbolische Informationsverarbeitung
- Graphentheoretische Probleme
- Planungsprobleme
- Rapid Prototyping
- ...

Entwicklung von Prolog

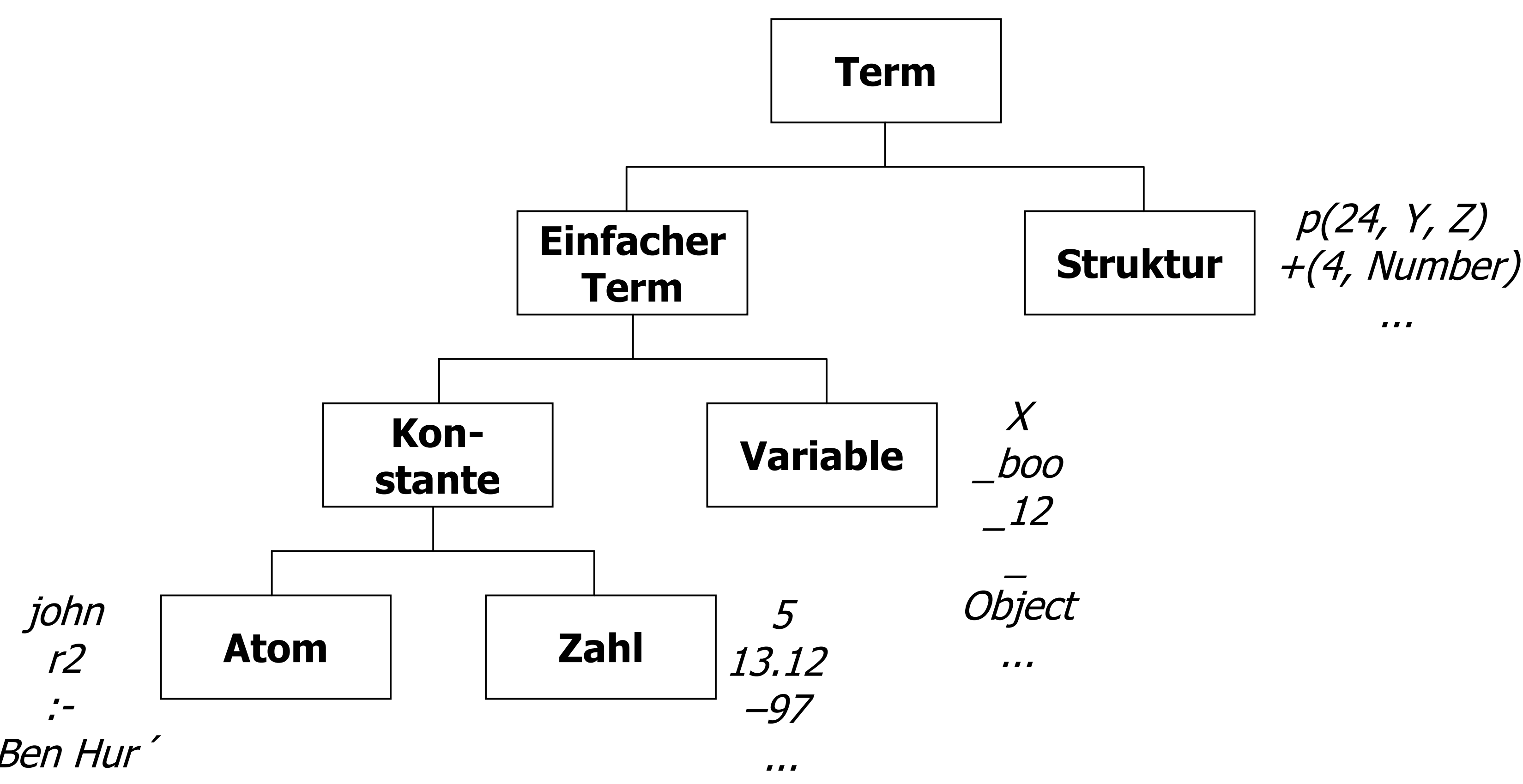
Einige Worte zur Entstehung von Prolog ...

Prolog-Versionen

Einige Worte zu den verschiedenen Prolog-Versionen
und –Dialekten ...

Syntax
Terme

Prolog-Programme sind aus Termen gebildet.



Syntax
Konstanten: Atome

Atome werden benötigt, um bestimmte Objekte oder bestimmte Beziehungen zu benennen.

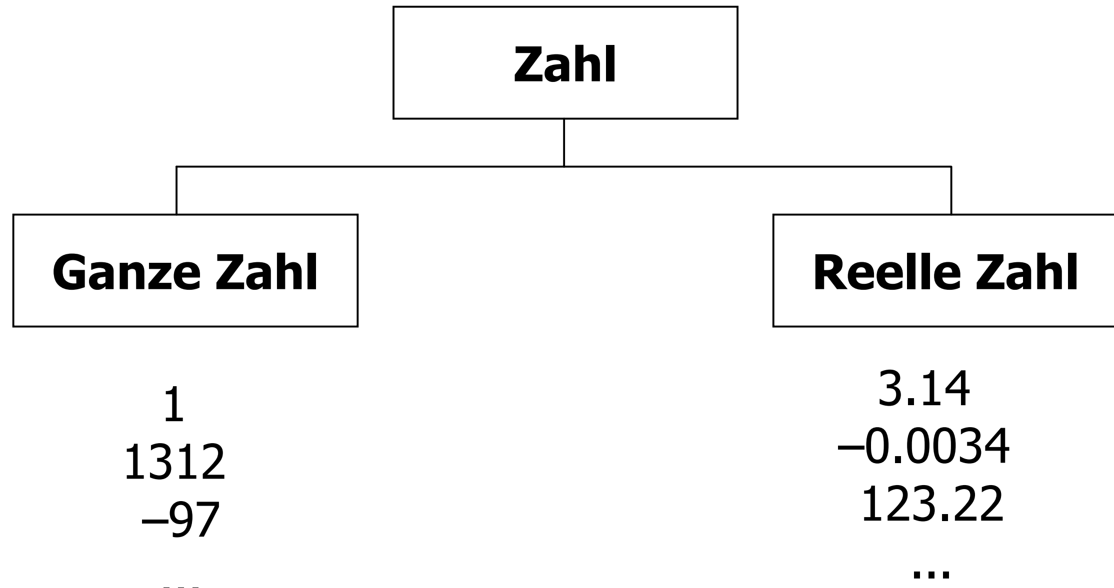
1. **Strings aus speziellen Characters**
   Beispiele: ?- :- + .:..

2. **Strings aus Buchstaben, Ziffern und dem *Underscore* Character, beginnend mit einem Kleinbuchstaben**
   Beispiele: susie is_person r2

3. **Strings aus beliebigen Characters, eingeschlossen in *Single Quotes***
   Beispiele: ´Bob Dole´ ´bill-1´ ´Calvin´

Characters	
Grossbuchstaben:	**A, B, ..., Z**
Kleinbuchstaben:	**a, b, ... , z**
Ziffern:	**0, 1, ... 9**
Spezielle Character:	**+ - * / \ ~ ^ < > : . ? @ # $ &**

Syntax
Konstanten: Zahlen



Reelle Zahlen werden in der Prolog-Programmierung selten verwendet, da Prolog eher für symbolische, nicht-numerische Berechnungen verwendet wird

Syntax
Variablen

Variablen sind Strings aus Buchstaben, Ziffern und dem *Underscore* Character, beginnend mit einem Grossbuchstaben oder mit einem *Underscore* Character. Variablen gelten innerhalb einer Klausel.

Beispiele: Answer_1 _ X Object _23

Anonyme Variable

Wenn eine Variable in einer Klausel nur einmal erscheint, muss sie nicht gebunden werden. Ihr Name ist daher irrelevant.

Kommen mehrere anonyme Variablen in derselben Klausel vor, müssen daher auch keine konsistene Interpretaionen für die Variablen gefunden werden.

```
is_mother(X) :- mother(X, Y).
```

äquivalent zu

```
is_mother(X) :- mother(X, _).
```

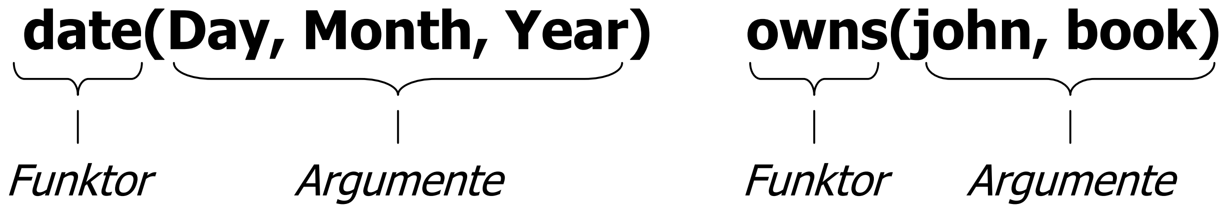
```
is_parent(X) :- mother(X, Y), father(X, Z).
```

äquivalent zu

```
is_parent(X) :- mother(X, _), father(X, _).
```

Syntax
Strukturen

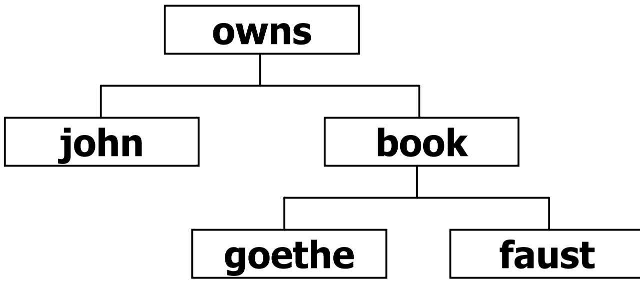
Strukturen sind Objekte, die aus Komponenten bestehen.



Die Komponenten können selber wieder Strukturen sein.

owns(john, book(goethe, faust))

Strukturen können als Bäume dargestellt werden.



Syntax
Matching

Die wichtigste Operation auf Prolog-Termen ist das *matching*.

Das *matching* zweier Terme ist erfolgreich, falls

1. die Terme identisch sind
2. die Variablen in beiden Termen durch Objekte instantiiert werden können, so dass die Terme nach der Substitution der Variablen durch diese Objekte identisch sind

Term1	Term2	Matching ...
date(Day, Month, 2001)	date(24, M, Y)	... succeeds
point(X, Y)	point(3, 5)	... succeds
plus(2, 2)	4	... fails
X	human(socrates)	... succeeds

Syntax: Matching
Übung

Ist das *matching* der folgenden Terme erfolgreich?

drinks(anna, wine)	drinks(Whom, What)
drinks(anna, Beverage)	drinks(Person, water)
book(14, goethe, X)	book(A, B, Y)
f(a, b, X)	f(a, c, X)
f(X, Y)	f(P, P)
cd(12, k(1685, 1750), bach)	cd(X, Y, _)
f(X, Y)	g(X, Y)

Syntax: Matching
Übung (Lösungen)

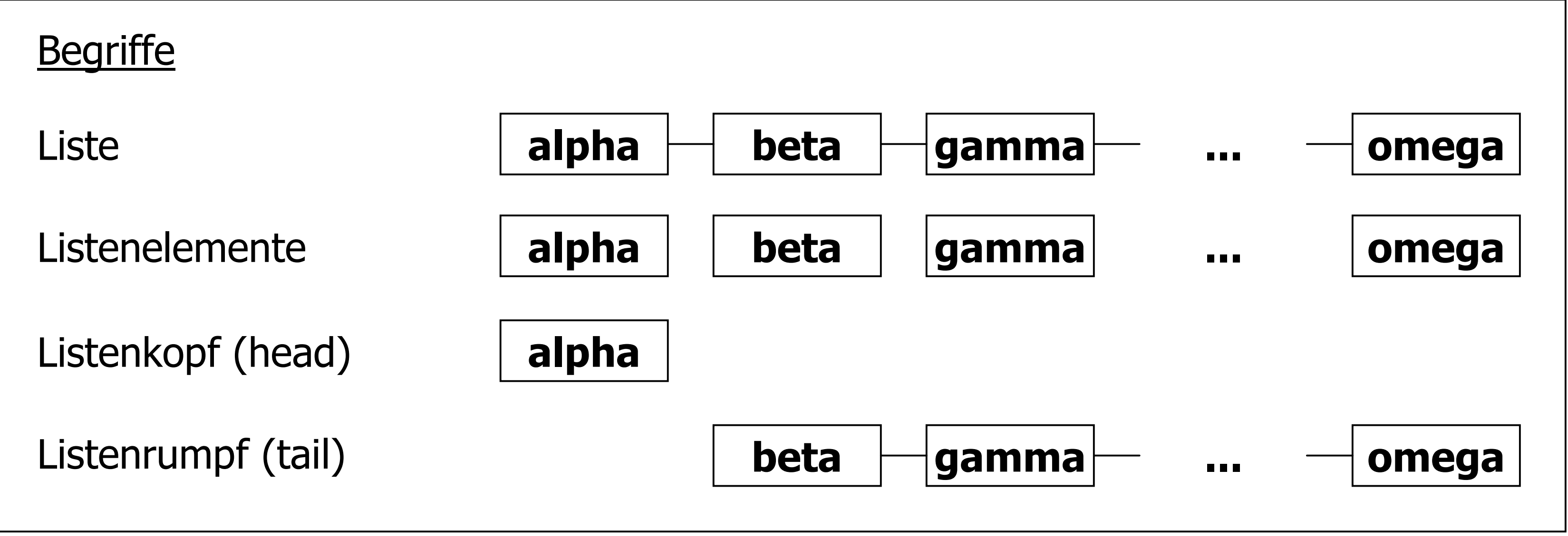
Ist das *matching* der folgenden Terme erfolgreich?

Term 1	Term 2	Lösung	
drinks(anna, wine)	drinks(Whom, What)	*ja:*	*Whom = anna,* *What = wine*
drinks(anna, Beverage)	drinks(Person, water)	*ja:*	*Person = anna,* *Beverage = water*
book(14, goethe, X)	book(A, B, Y)	*ja:*	*A = 14, B = goethe* *X = Y*
f(a, b, X)	f(a, c, X)	*nein:*	$b \neq c$
f(X, Y)	f(P, P)	*ja:*	*X = P, Y = P (→ X = Y)*
cd(12, k(1685, 1750), bach)	cd(X, Y, _)	*ja:*	*X = 12* *Y = k(1685, 1750)*
f(X, Y)	g(X, Y)	*nein*	$f \neq g$

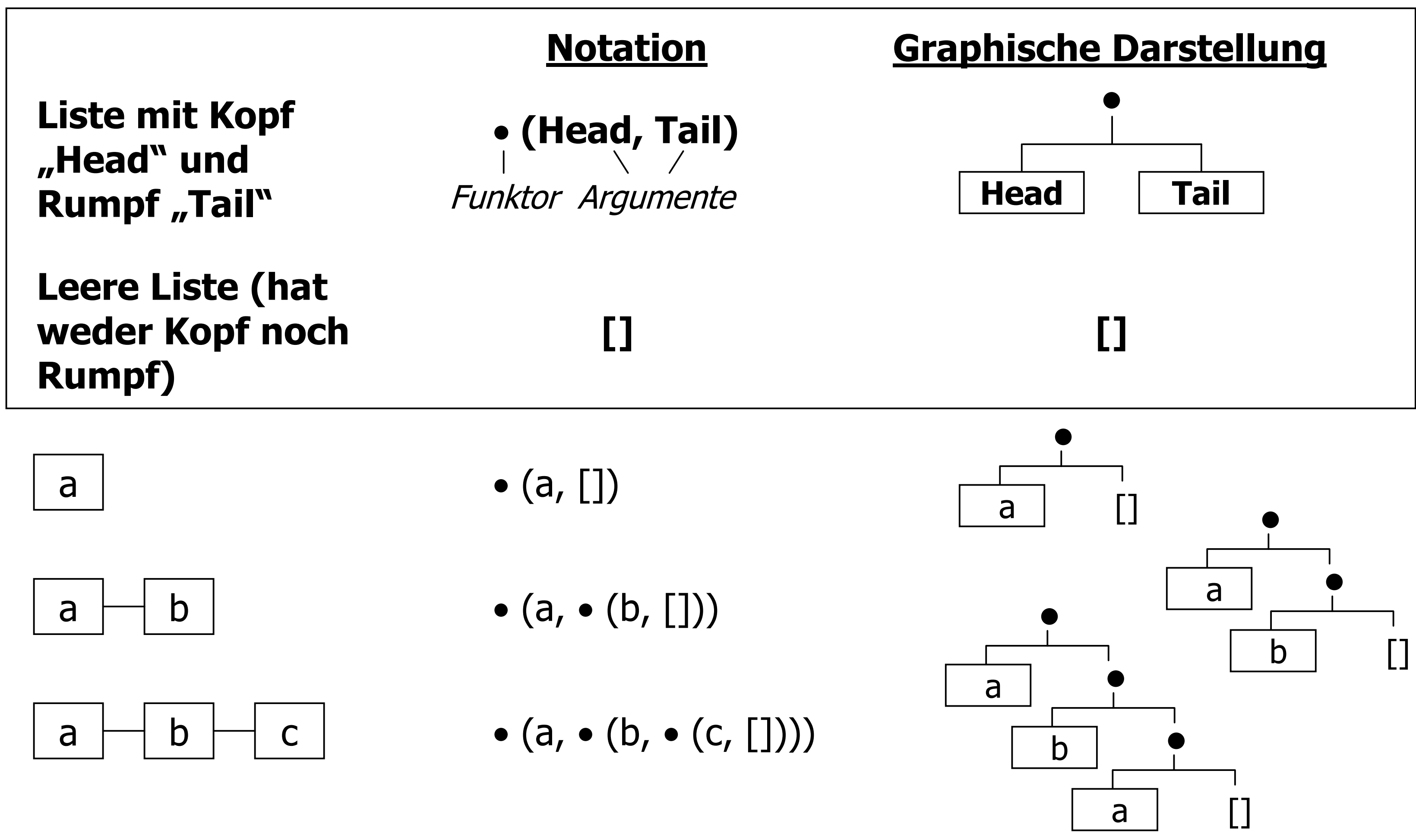
Listen

Liste: - beliebig lange Sequenz von Listenelementen
- syntaktisch sind Listen Strukturen

Listenelemente: Terme (→ können auch selbst wieder Listen sein)



Repräsentation von Listen
Funktor-Notation



Repräsentation von Listen
Listen-Notation

	Notation 1	Notation 2
a	**[a]**	**[a|[]]**
a — b	**[a, b]**	**[a|[b]]**
a — b — c	**[a, b, c]**	**[a|[b, c]]**

Beispiele

[[the, man], walks]	=	[[the, man]|[walks]]
[the, man, [walks, fast]]	=	[the|[man, [walks, fast]]]
[the, [man, walks, fast]]	=	[the|[[man, walks, fast]]]
[]	=	[]

Repräsentation von Listen
Übung

1. Ist das *matching* der folgenden Listen erfolgreich? Falls ja, wie werden die Variablen instanziiert?

[man]	[X|Y]
[[X, cat], [is, cute]]	[[the, Y]|Z]
[united, kingdoms]	[united|Tail]

2. Gegeben ist folgendes Fakt: p([H|T], H, T).
   Welche Antworten erfolgen auf die folgenden Fragen?

   ?- p([2, 4, 8], X, Y).

   ?- p([sunny], X, Y).

   ?- p([[a, b, c], [A, B, C]], X, Y).

Repräsentation von Listen
Übung (Lösungen)

1. Ist das *matching* der folgenden Listen erfolgreich? Falls ja, wie werden die Variablen instanziiert?

[man]	[X|Y]	*ja: X = man, Y = []*
[[X, cat], [is, cute]]	[[the, Y]|Z]	*ja: X = the, Y = cat, Z = [[is, cute]]*
[united, kingdoms]	[united|Tail]	*ja: Tail = [kingdoms*]

2. Gegeben ist folgendes Fakt: p([H|T], H, T).
   Welche Antworten erfolgen auf die folgenden Fragen?

?- p([2, 4, 8], X, Y).	*X = 2, Y = [4, 8]*
?- p([sunny], X, Y).	*X = sunny, Y = []*
?- p([[a, b, c], [A, B, C]], X, Y).	*X = [a, b c], Y = [[A, B, C]]*

Operationen auf Listen
Operation *member*

```
% member(X, L) : X ist ein Element der Liste L
member(X, [X|Tail]).
member(X, [Head|Tail] :- member(X, Tail).
```

?- member(beta, [alpha, beta, gamma, delta]).
yes

Operationen auf Listen
Operation *append*

```
% append(X, Y, Z) :  Z ist die Liste, die entsteht, wenn man die
%                    Elemente der Liste Y der Reihe nach hinten
%                    an die Liste X anhängt (Konkatenation)
append([], L, L).
append([X|L1], L2, [X|L3] :- append(L1, L2, L3).
```

?- append([a, b], [c, d], U).

U = [a, b, c, d].

Operationen auf Listen
Übung

Definieren Sie die folgenden Prädikate:

% prefix(P, L) : P ist ein Präfix von L
Beispiel: prefix([a, b], [a, b, c, d]).

% last(X, L) : X ist das letzte Element von L
Beispiel: last(c, [a, b, c]).

% reverse(L1, L2) : L2 ist die invertierte Liste von L1
Beispiel: reverse([a, b, c], [c, b, a]).
Tip: Verwenden Sie die Prozedur append

Operationen auf Listen
Übung (Lösungen)

```
% prefix(P, L) : P ist ein Präfix von L
prefix([], _).
prefix([X|R], [X|S]) :- prefix(R, S).

% last(X, L) : X ist das letzte Element von L
last(X, [X]).
last(X, [_|Y]) :- last(X, Y).

% reverse(L1, L2) : L2 ist die invertierte Liste von L1
reverse([], []).
reverse([X|L], M) :- reverse(L, N), append(N,[X], M).
```

Layout von Prolog-Programmen

- Alle Klauseln einer Prozedur unmittelbar hintereinander schreiben
- Eine Leerzeile zwischen zwei Prozeduren einfügen
- Jede Klausel auf einer neuen Zeile beginnen
- *Falls genügend Platz vorhanden ist:* Jede Klausel auf eine eigene Zeile schreiben
  Sonst: Den Kopf und das Zeichen :- auf die erste Zeile und jedes zu beweisende Ziel (eingerückt) auf eine neue Zeile schreiben
- Alle Prozeduren kurz dokumentieren

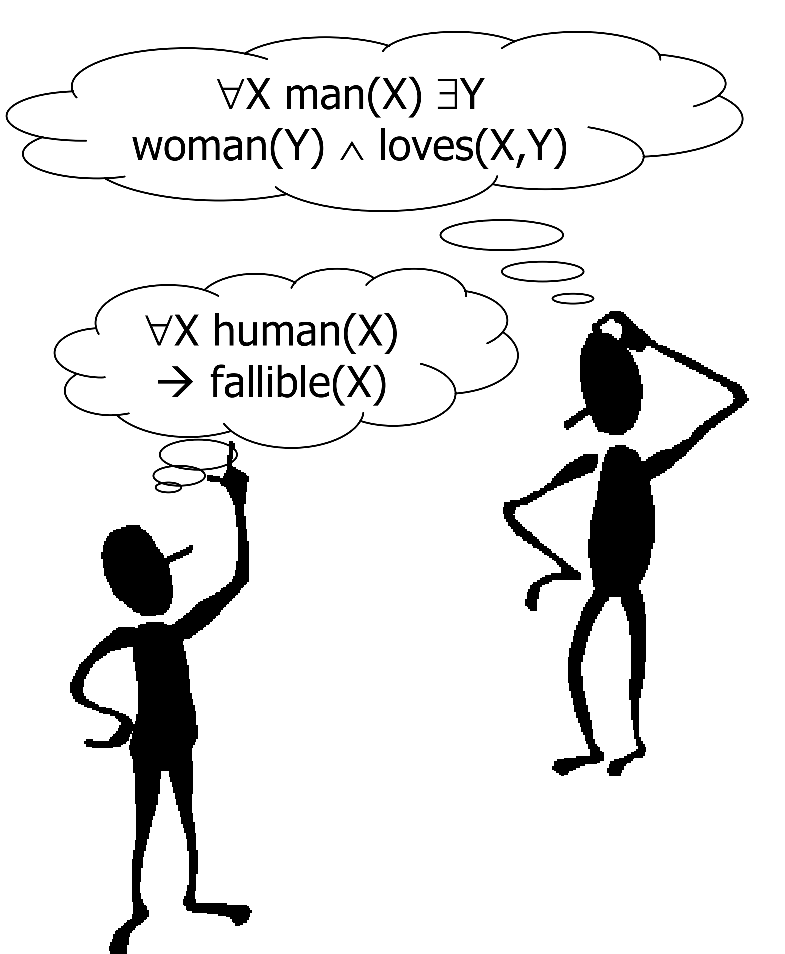
  % Dies ist ein einzeiliger Kommentar

  /* Dies könnte auch ein mehrzeiliger Kommentar sein */

Logische Basis von Prolog
Logik

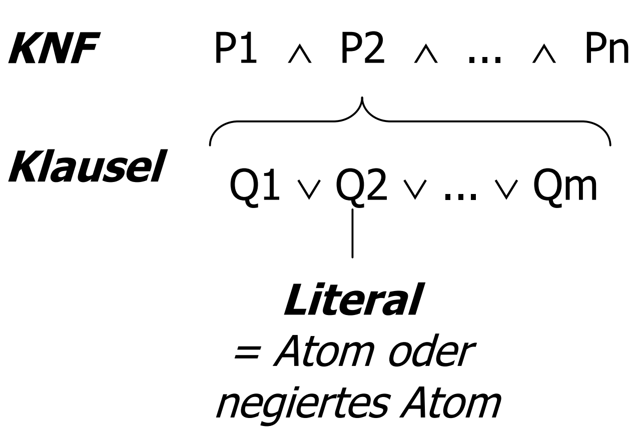
- Logik ist die Lehre vom folgerichtigen Denken.
- Die Sprache der Logik kann verwendet werden, um gewissen Aussagen Wahrheitswerte zuzuordnen.
- Aussagen können miteinander verknüpft werden.
- Neue Aussagen können aus vorhandenen Aussagen hergeleitet werden.

Logische Basis von Prolog
Prädikatenlogik



- Die gebräuchlichste Form der Logik ist die sogenannte Prädikatenlogik.
- In der Prädikatenlogik werden Aussagen mit Hilfe von *und* (∧), *oder* (∨), *nicht* (¬), *wenn* (→), und *genau dann wenn* (↔), verknüpft.
- Aussagen können Variablen verwenden. Diese müssen durch einen Quantor eingeführt werden (∀: Allquantor, ∃: Existenzquantor).

Logische Basis von Prolog
Konjunktive Normalform und Klauseln



<u>Beispiel (einer Klausel):</u>

$\neg$human(X) $\vee$ fallible(X)

negatives Literal — *positives Literal*

Formeln der Prädikatenlogik können vereinfacht werden:

- Die Quantoren werden entfernt (alle Variablen werden als implizit allquantifiziert betrachtet).
- Nur noch die elementaren Verknüpfungen *und*, *oder* und *nicht* kommen vor.
- Die Verknüpfungen *und*, *oder* und *nicht* sind von aussen nach innen sortiert.

Diese Form heisst *konjunktive Normalform (KNF).* Die durch *und* verknüpften Elemente heissen *Klauseln*.

Logische Basis von Prolog
Hornklauseln

Hormklausel mit *einem* positiven Literal:

$$\neg \text{human}(X) \vee \text{fallible}(X)$$

$$\leftrightarrow$$

$$\text{fallible}(X) \leftarrow \text{human}(X)$$

Hormklausel mit *keinem* positiven Literal:

human(socrates)

- Im Programmieren in Logik beschränkt man sich auf eine eingeschränkte Klauselform: auf *Hornklauseln*.
- Hornklauseln sind Klauseln mit höchstens einem positiven Literal.

Logische Basis von Prolog
Inferenzregeln

Beispiele von Inferenzregeln

$a, a \rightarrow b$
---------- *Modus Ponens*
b

a, b
---------- *Und-Einführung*
$a \wedge b$

Enthält eine Formelmenge die beiden Formeln über dem Strich, so darf auch die Formel unter dem Strich hinzugefügt werden.

- Eine Inferenzregel gibt an, wie aus einer Menge von Formeln eine neue Formel abgeleitet werden kann.
- Eine Menge von Inferenzregeln ist *korrekt*, wenn jede hergeleitete Formel inhaltlich korrekt ist.
- Eine Menge von Inferenzregeln ist *vollständig*, wenn jede inhaltlich korrekte Formel hergeleitet werden kann.

Logische Basis von Prolog
Mechanisches Beweisen und Resolution

Resolution

$a \vee b1 \vee b2 \vee ... \vee bn,$
$\neg a \vee c1 \vee c2 \vee ... \vee cm$

$b1 \vee ... \vee bn \vee c1 \vee ... \vee cm$

- In der logischen Programmieren soll der Beweisprozess mechanisch erfolgen.
- Dabei stellt sich die Frage:

  Welche Inferenzregel soll auf welche Formel angewendet werden?
- Die Resolution ist eine Inferenz-regel, die vollständig und korrekt ist.
- Es genügt daher, nur die Inferenz-regel *Resolution* zu verwenden.
- Die Resolution operiert auf Klauseln.

Logische Basis von Prolog
Logik und Prolog

Prolog-Programm

=

Logik

+

Steuerung

- Fakten, Regeln und Fragen sind Hornklauseln
- Um eine Frage zu beweisen, benützt Prolog eine spezielle Form der Resolution: die sogenannte SLD-Resolution (**S**elect **L**inear **D**efinite Clauses-Resolution).
- Bei der Auswahl der Klauseln benützt die SLD-Resolution eine bestimmte Auswahlstrategie. Diese ist nicht „fair". Der Interpreter kann in Endlosschleifen gelangen.
- Prolog-Interpreter sind daher nicht vollständig.

Built-In-Prädikate

Input/Output-Prädikate	*z. B. um einen Term einzulesen oder auszugeben*
Arithmetische Prädikate	*z. B. um eine Multiplikation auszuführen*
Wissensbasis Prädikate	*z. B. um der Wissensbasis zur Laufzeit Fakten hinzuzufügen oder zu entfernen*
Struktur-Prädikate	*z. B. um zu testen, ob ein Term ein Atom ist*
Ausführungs-Kontroll-Prädikate	*z. B. um den Bearbeitungsablauf innerhalb eines Programms zu verändern*
...	*...*

Built-In-Prädikate
Beispiele von Input/Output-Prädikaten

- **read(Term)** liest den nächsten Term vom aktuellen Eingabe-Medium. Der Term muss mit einem Punkt gefolgt von einem Leerzeichen abgeschlossen werden.
- **write(Term)** schreibt das Argument *Term* in das aktuelle Ausgabemedium.
- **nl** schreibt einen Zeilenvorschub in das aktuelle Ausgabemedium.
- Die Prädikate **read** und **write** können nur einmal erfüllt werden.

Beispiele

```
read_and_write :-
        read(T), write(T), nl.

?- read_and_write.
amen.
amen
yes
------------
writelist([]).
writelist([X|L]) :-
        write(X), nl, writelist(L).

?- writelist([one, two]).
one
two
yes
```

Built-In-Prädikate
Beispiele arithmetischer Prädikate

- **X is ArithmeticExpression** ist erfüllt, wenn X gleich dem arithmetischen Wert von *ArithmeticExpression* ist.

  is *wertet den arithmetischen Ausdruck nach den üblichen Regeln aus und weist das Ergebnis der Variablen X zu oder vergleicht es mit dem numerischen Wert, auf den X instanziiert ist*.

- nicht verwechseln mit dem Vergleichsprädikat **= :**

  X = Y ist erfüllt, wenn X und Y unifiziert werden können.

Beispiele

```
?- A = 3+4.
A = 3+4;
no

?- A is 3+4.
A = 7;
no
------------
factorial(0, 1).
factorial(N, R) :-
        N > 0,
        N1 is N-1,
        factorial(N1, R1),
        R is N*R1.
```

Built-In-Prädikate
Übung

Wie antwortet Prolog auf die folgenden Fragen?

```
?- A is 3, B is A + 4.
?- A = 3, B is A + 4.
?- A = aha, B = A + 4.
?- A = aha, B is A + 4.
?- A is 3, B = A + 4.
```

Built-In-Prädikate
Übung (Lösung)

?- *A is 3, B is A + 4.*
A = 3
B = 7;
no

?- *A = 3, B is A + 4.*
A = 3
B = 7;
no

?- *A = aha, B = A + 4.*
A = aha
B = aha + 4;
no

?- *A = aha, B is A + 4.*
no

?- *A is 3, B = A + 4.*
A = 3
B = 3+4;
no

Built-In-Prädikat „Cut“
Beispiel: Maximum zweier Zahlen

max(X, Y, Max) ist wahr, wenn Max das Maximum von X und Y ist.

Variante 1

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

Wenn die erste Regel zum Erfolg führt, dann ist klar, dass die zweite Regel nicht mehr zum Erfolg führen kann.

Variante 2

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

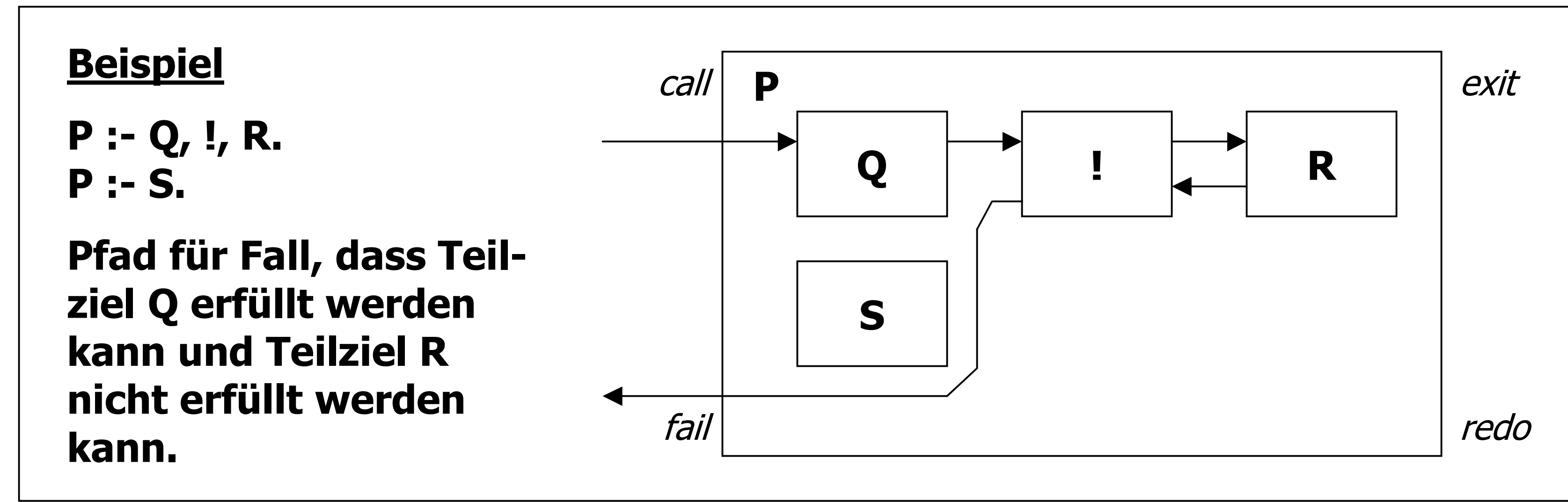
Durch Hinzufügen des Cut-Operators kann unnötiges Backtracking verhindert werden.

Text: Clocksin/Mellish, S.75-79

Text austeilen und Zeit zum Lesen geben ...

Built-In-Prädikat „Cut“
Semantik

- Notation: !
- Wird der ! das erste Mal als zu beweisendes Ziel angetroffen, so gilt er unmittelbar als bewiesen.
- Sollen später im Rahmen eines Backtrackings Ziele bewiesen werden, die vor dem ! liegen, so verhindert der ! dies.



```
facility(Pers, Fac) :- book_overdue(Pers, Book), !, basic_facility(Fac).
facility(Pers, Fac) :- general_facility(Fac).

basic_facility(reference).
basic_facility(enquiries).

additional_facility(borrowing).
additional_facility(inter_library_loan).

general_facility(X) :- basic_facility(X).
general_facility(X) :- additional_facility(X).

book_overdue(‚C. Watzer‘, book10089).
book_overdue(‚A. Jones‘, book29907).
...

client(‚A. Jones‘).
client(‚W. Metesk‘).
...
```

Beispiel zum „Cut"

Quelle: W. F. Clocksin und C. S. Mellish, Programming in Prolog, 4th Edition, Springer Verlag, 1994, S.76 (Kapitel 4.2 The „Cut")

Built-In-Prädikat „Cut“
Übung

Gegeben sei das folgende Prolog-Programm:

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

Was antwortet das Prolog-System auf die folgenden Fragen?

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

Built-In-Prädikat „Cut“
Übung (Lösungen)

Gegeben sei das folgende Prolog-Programm:

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

Was antwortet das Prolog-System auf die folgenden Fragen?

?- p(X).	*X=1; X=2*
?- p(X), p(Y).	*X=1, Y=1; X=1, Y=2; X=2, Y=1; X=2, Y=2*
?- p(X), !, p(Y).	*X=1, Y=1; X=1, Y=2*

Built-In-Prädikat „Cut"

Negation durch „Cut & Fail"-Kombination

Prolog-Klauseln erlauben nur die Darstellung positiver Information.

Wie können z.B. die folgenden Sachverhalte dargestellt werden?

- Anna ist keine Studentin.
- Anna mag alle Tiere ausser Schlangen.

Annahme: Was nicht hergeleitet werden kann, ist falsch (Closed World Assumption).

```
not_student(X) :- student(X), !, fail.
not_student(X).

likes(anna, X) :- snake(X), !, fail.
likes(anna, X) :- animal(X).
```

fail *ist ein Prädikate das nicht erfüllt ist und daher Backtracking verursacht.*

Built-In-Prädikat „Cut"
Vorsicht!

- Es kann sein, dass ein ! richtige Resultate produziert, wenn die Fragen eine bestimmte Form haben. Dies garantiert aber nicht, dass die Prozedur auch für Fragen mit einer anderen Form richtig funktioniert.
- Der ! kann die deklarative Bedeutung einer Prozedur verändern: Wenn die Reihenfolge der Klauseln geändert wird, kann sich die Bedeutung der Prozedur ändern.
- ...

→ *Bei der Verwendung von ! vorsichtig sein!*

Built-In-Prädikat „Cut“
Vorsicht! (Beispiel 1)

max(X, Y, Max) ist wahr, wenn Max das Maximum von X und Y ist.

Variante 2

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

Variante 3

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

Variante 4

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

Vorsicht!

Variante 3 funktioniert nur, wenn die Variable Max im zu beweisenden Ziel max(X, Y, Max) nicht instantiiert ist.

Die Anfrage max(4, 1, 1) beispielsweise führt zu einer falschen Antwort.

Variante 4 eliminiert dieses Problem, indem erst nach dem Cut geprüft wird, ob das erste und das dritte Argument gleich sind.

Built-In-Prädikat „Cut“

Vorsicht! (Beispiel 2)

Ohne !: Reihenfolge der Klauseln beeinflusst deklarative Bedeutung nicht

```
p :- a, b.        p :- c.
p :- c.           p :- a, b.
```

$p \leftrightarrow (a \wedge b) \vee c \quad \longleftrightarrow \quad p \leftrightarrow c \vee (a \wedge b)$

Mit !: Reihenfolge der Klauseln beeinflusst deklarative Bedeutung

```
p :- a, !, b.     p :- c.
p :- c.           p :- a, !, b.
```

$p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c) \quad \not\longleftrightarrow \quad p \leftrightarrow c \vee (a \wedge b)$

Anwendungen

Einige konkrete Anwendungen besprechen ...

Logische Programmierung und Prolog
Literatur

- W. F. Clocksin und C. S. Mellish, *Programming in Prolog*, 4th Edition, Springer Verlag, 1994.
- I. Bratko, *Prolog: Programing for Artificial Intelligence*, Third Edition, Addison Wesley, 2001.
- L. Sterling und E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 2nd Edition, 1994.