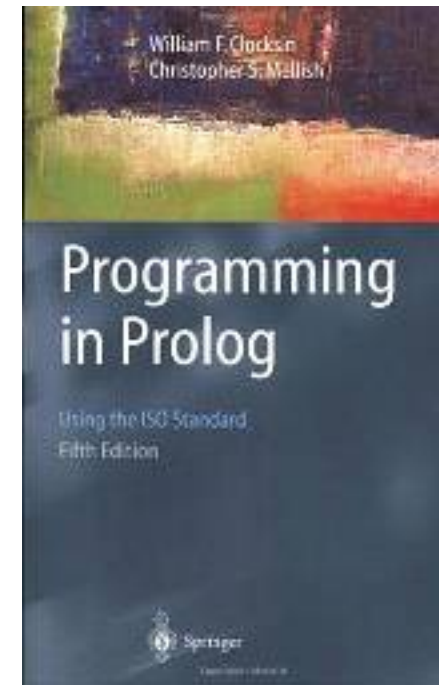


# 4. Logikorientierte Programmiersprachen

- 1 Grundlagen
- 2 Funktionale Programmiersprachen
- 3 Verarbeitung von Sprache
- 4 Logikorientierte Programmiersprachen
  - a. Einführung in Prolog
  - b. Unifikation, Backtracking
  - c. Anwendungsbeispiele von Prolog
  - d. Prolog und Logik
  - e. Vorwärtsverkettung mit Drools

# Literatur

- W.F. Clocksin, C.S. Mellish: Programming in Prolog: Using the ISO Standard, Springer Berlin Heidelberg, 2013
- SWI Prolog Download
  - <http://www.swi-prolog.org>
- SWI Prolog Online
  - <http://swish.swi-pro>
  - Nur eine Teilmenge von SWI Prolog
- Prolog Kurs
  - <http://www.learnprolognow.org>



# Prolog

- Prolog = **P**rogrammieren in **L**ogik
  - Wichtigste logische Programmiersprache
  - Deklaratives Programmieren
  - Es wird beschrieben, WAS das Problem ist, nicht WIE es gelöst wird
  - Fakten, Regeln und Anfragen
  - Auf built-in Prädikate wird weitgehend verzichtet
- Anwendungsbereiche
  - Künstliche Intelligenz, Computerlinguistik, Expertensysteme
- Interpreter
  - Verwendung von SWI-Prolog (GNU Public License)
  - <http://www.swi-prolog.org>
  - Interface zu C, C++, ODBC, ...

# Fakten

- Klauseln
  - Fakten, z.B. Elsa ist eine Kuh
  - Regeln, z.B. Kühe fressen Grass
- Fakten
  - Definition durch n-stellige Prädikate
  - Mary is female: `female(mary)`.
  - John likes Mary: `likes(john, mary)`.
  - John gives Mary the book: `gives(john, mary, book)`.
- Definition von Fakten
  - Verwendung von Kleinbuchstaben
  - Punkt am Ende eines Fakts
  - Fakten gleicher Prädikate sollten zusammen angegeben werden

# Anfragen / Queries

- Syntax für Anfragen: ? –
- Beispiel
  - ?– female(joe).
  - Antwort: yes oder no
- Abfragen mit Variablen
  - ?– female(X).
  - Antwort: X=mary
  - Weitere Ergebnisse durch Semikolon ;
- Ergebnis einer Anfrage
  - Falls Anfrage Variablen enthält: gefundene Belegungen oder no
  - Falls Ergebnis keine Variablen enthält: yes oder no

# Konjunktion

- Operator Komma „ , “ für Konjunktion
- Beispiel
  - `likes(mary, food).`  
`likes(mary, wine).`  
`likes(john, wine).`  
`likes(john, mary).`
  - Anfrage:  
`?- likes(john, mary), likes(mary, john).`
- Beispiel 2
  - Gibt es etwas, das John und Mary gemeinsam mögen?
  - `?- likes(john, X), likes(mary, X).`
  - Auswertung dieser Anfrage: finde heraus, welche X John mag und finde heraus, ob Mary dieses X auch mag

# Regeln

- Regeloperator : –
  - Umgekehrter Implikationspfeil
  - Instanziierung von Variablen durch einzelne Werte
- Beispiel 1
  - John mag alle Frauen, die Wein mögen
  - `female(mary).`
  - `likes(mary, food).`
  - `likes(mary, wine).`
  - `likes(john, wine).`
  - `likes(john, X) :- female(X), likes(X, wine).`
  - Was ist das Ergebnis dieser Anfrage?
  - Wie funktioniert die Auswertung dieser Anfrage?

# Vorteile Logik-Programmierung

- Ein Prädikat erfasst mehrere Fälle, für die in imperativen Programmiersprachen unterschiedliche Funktionen notwendig wären
- Beispiele
  - Mag Mary Wein?  
`likes(mary, wine).`
  - Wer mag Wein?  
`likes(X, wine).`
  - Was mag Mary?  
`likes(mary, X).`
  - Wer mag was?  
`likes(X, Y).`



# Terme in Prolog

- Konstante
  - Konstante beginnen mit einem Kleinbuchstaben
- Variable
  - Variablen fangen mit Großbuchstaben oder Unterstrich „\_“ an
- Strukturen
  - `owns(john, book).`
  - `owns` wird hier Funktor genannt
  - `john` und `book` sind Komponenten
  - Komponenten können Variable, Konstante oder Zahlen sein
- Strukturen können verschachtelt sein
  - `owns(john, book(moby_dick, author(herman, melville))).`

# Disjunktion

- Definition mehrerer Regeln zur Definition einer Disjunktion
  - `grossvater(X,Y) :- vater(X,Z), vater(Z,Y).`
  - `grossvater(X,Y) :- vater(X,Z), mutter(Z,Y).`
- Alternative durch Verwendung Semikolon-Operator ;
  - `grossvater(X,Y) :- (vater(X,Z), vater(Z,Y))`  
`;`  
`(vater(X,Z), mutter(Z,Y)).`

# Regeln

- Beispiel 2

- `male(albert).`  
`male(edward).`  
`female(alice).`  
`female(victoria).`  
`parents(edward, victoria, albert).`  
`parents(alice, victoria, albert).`
- `sister_of(X,Y) :- female(X),`  
`parents(X,M,F),`  
`parents(Y,M,F).`
- `?- sister_of(alice, Z).`
- Was ist das Ergebnis diese Anfrage?
- Wie funktioniert die Auswertung dieser Anfrage?

# Regeln

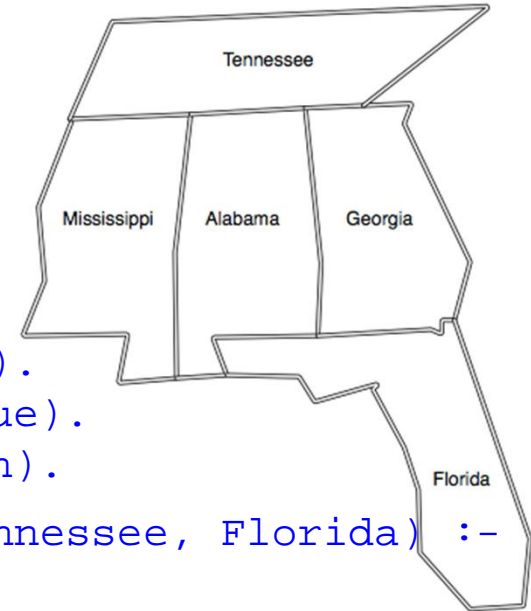
- Beispiel 3

- `female(mary).`  
`thief(john).`  
`likes(john, wine).`  
`likes(mary, food).`  
`likes(mary, wine).`  
`likes(john, X) :- female(X), likes(X, wine).`  
`may_steal(X, Y) :- thief(X), likes(X,Y).`
- `?- may_steal(john, X).`
- Was ist das Ergebnis diese Anfrage?
- Wie funktioniert die Auswertung dieser Anfrage?
- `?- may_steal(X, wine).`
- `?- may_steal(X, Y).`

# Regeln

- Beispiel: Einfärben einer Landkarte

- `different(red, green). different(red, blue).`  
`different(green, red). different(green, blue).`  
`different(blue, red). different(blue, green).`
- `coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-`  
`different(Mississippi, Tennessee),`  
`different(Mississippi, Alabama),`  
`different(Alabama, Tennessee),`  
`different(Alabama, Georgia),`  
`different(Alabama, Florida),`  
`different(Georgia, Florida),`  
`different(Georgia, Tennessee).`
- `?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).`
- Was ist die Ausgabe, in welcher Reihenfolge werden die Lösungen ausgegeben?
- Wie kann die Definition von `different` verkürzt werden?



Quelle: B.A. Tate: Sieben Wochen, sieben Sprachen, O'Reilly

# Equality und Arithmetic

- Equality
  - $?- X = Y.$
  - Aussprache: equals
  - Wenn X uninstantiiert und Y instantiiert, wird X zu Y instantiiert
  - Wenn beides Atome sind, wird Vergleich durchgeführt
  - Zwei Strukturen sind gleich, wenn Funktor und Komponenten gleich
- Prädikat ==
  - Liefert nur dann true zurück, wenn beide Terme bereits identisch sind
  - Es wird keine Unifikation durchgeführt
  - $X == 4$  ist nur dann true, wenn X bereits 4 ist
- Prädikate \= bzw. \==
  - Ergeben false, wenn die entsprechenden Gleichheitsoperatoren true ergeben

# Equality und Arithmetic

- Arithmetic
  - Prädikate = , \=, <, >, =<, >=
  - Operator is: Y is P/A
  - Werte auf rechten Seite müssen bekannt sein
- Built-in-Prädikate und Built-in-Operatoren
  - Input/Output: write(<term>) read(<term>)
- Kommentare
  - Kommentar zwischen /\* und \*/  
/\* Kommentar \*/
  - Kommentierung Rest der Zeile  
% Kommentar

# Equality und Arithmetic

- Beispiel zu Operator `is`

- `pop(usa, 203).`  
`pop(india, 548).`  
`pop(china, 800).`  
`pop(brazil, 108).`
- `area(usa, 3).`  
`area(india, 1).`  
`area(china, 4).`  
`area(brazil, 3).`
- `density(X, Y) :-`  
`pop(X, P),`  
`area(X, A),`  
`Y is P/A.`

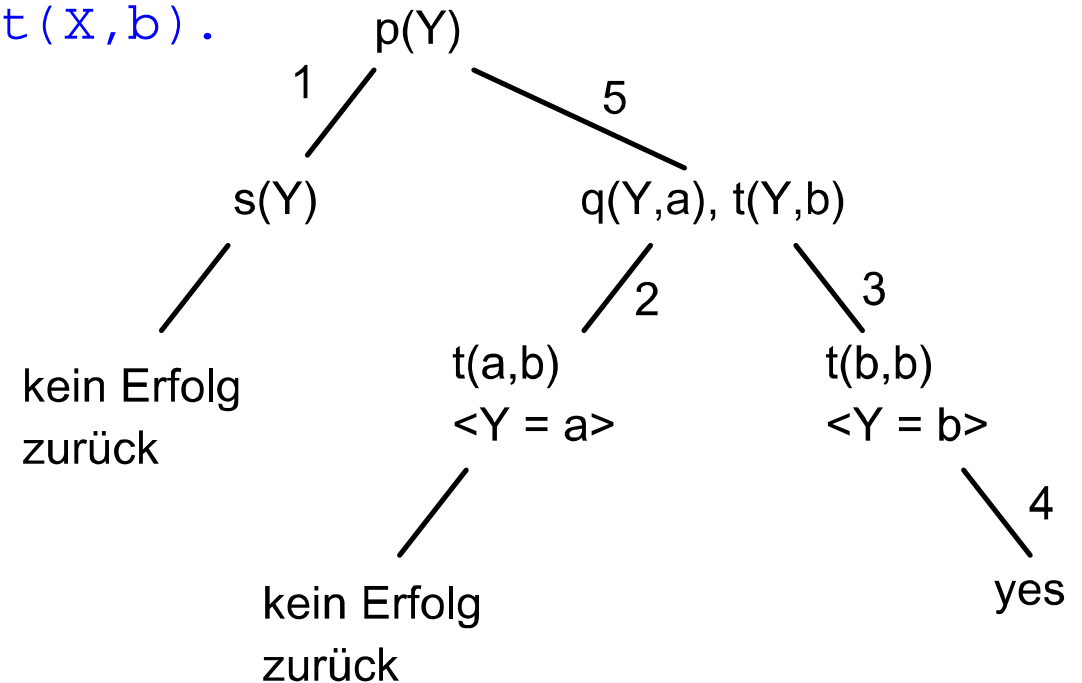


# Lösungssuche in Prolog

- Beispiel

```
1. p(X) :- s(X).  
2. q(a,a).  
3. q(b,a).  
4. t(b,b).  
5. p(X) :- q(X,a), t(X,b).
```

```
?- p(Y).
```



# Unification (Unifikation, Unifizierung)

- Unification bedeutet Gleichsetzung („matching“)
  - Variable mit Integer, Atome: Variable wird gebunden
  - Variable mit Variable:
    - Falls eine Variable an Wert gebunden wird, ist automatisch auch andere gebunden
    - Falls beide Variablen nicht gebunden, dann wird die eine automatisch gebunden, wenn die andere gebunden wird („share“)
  - Variable mit Struktur
  - Atom und Integer untereinander: unifizierbar, falls Werte identisch
  - Struktur mit Struktur: falls gleicher Functor, gleiche Anzahl Argumente und alle Argumente matchen
    - $?- (A, B, 3) = (1, 2, C).$
    - $?- a(b, C, d(e, F, g(h, i, J)))$   
 $= a(B, c, d(E, f, g(H, i, j))).$

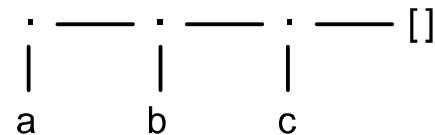
# Datenstrukturen

- Strukturen und Bäume

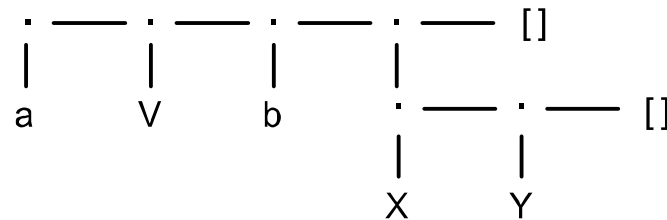
- Überführen von Strukturen in Bäume
- $a+b*c$ .
- `book(moby_dick, author(herman, melville)).`

- Listen

- Listen als spezielle Bäume
- Darstellung der Liste `[a,b,c]` als Baum: `.(a,.(b,.(c,[])))`



- Wie wird die folgende Liste dargestellt? `[a,v,b,[X,Y]]`



# Listen

- Aufteilung einer Liste
  - Split in Head und Tail:  $[X|Y]$
  - $p([1,2,3]).$   
   $?- p([X|Y]).$

Liste	Head	Tail
$[a,b,c]$	$a$	$[b,c]$
$[a,b]$	$a$	$[b]$
$[the,[cat,sat],down]$	$the$	$[[cat,sat],down]$
$[X+Y,x+y]$	$X+Y$	$[x+y]$

- $p([the,cat,sat,[on,the,mat]]).$
- $?- p([X|Y]).$   
   $?- p([_,_,_,[_|Y]]).$   
   $?- p([_,_,X|Y]).$

# Matching von Listen

- Beispiele

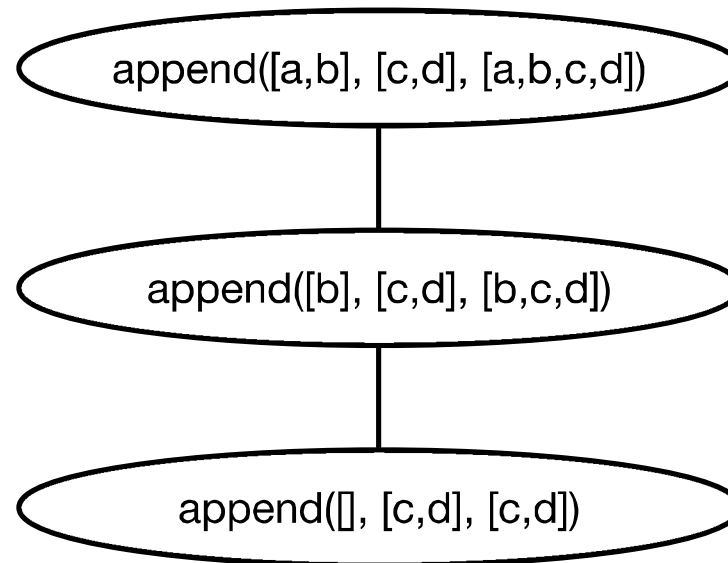
Liste 1	Liste 2	Instanziierung
<code>[X,Y,Z]</code>	<code>[john,likes,fish]</code>	
<code>[cat]</code>	<code>[X Y]</code>	
<code>[X,Y Z]</code>	<code>[mary,likes,wine]</code>	
<code>[[the,Y] Z]</code>	<code>[[X,hare],[is,here]]</code>	
<code>[golden T]</code>	<code>[golden,norfolk]</code>	
<code>[white,horse]</code>	<code>[horse,X]</code>	
<code>[white Q]</code>	<code>[P,horse]</code>	

# Rekursion

- Anonyme Variable `_`
  - Unifikation mit allem, Inhalt ist hier egal
  - Variable wird nicht ausgegeben
  - Vermeidung der Warnung "singleton variables: "
- Durchsuchen einer Liste
  - Rekursiver Algorithmus
  - `member(X, [X|_]).`
  - `member(X, [_|Y]) :- member(X, Y).`
  - Was ergibt folgende Anfrage? `?- member(c, [a, b, c]).`
- Aufgabe
  - Definieren Sie die Berechnung der Fakultät in Prolog.

# Rekursion bei Listen – Beispiel Append

- Funktion zum Konkatenieren zweier Listen
  - `?- append([a,b],[c,d], X).`  
`X = [a,b,c,d]`
- Lösungsidee



# Rekursion bei Listen - Beispiel

- Funktion zum Konkatenieren zweier Listen

- `append([], L, L).`
- `append([H|T1], L, [H|T2]) :- append(T1, L, T2).`
- `?- append([1,2],[3],W).`
- `W = [1,2,3].`

- Unifikation führt zu folgenden Teilzielen


- `append([1|[2]], [3], [1|T2-A]) :- append([2],[3], T2-A).`
- `append([2|[]], [3], [2|T2-B]) :- append([], [3], T2-B).`
- `append([], [3], [3]).`

- Was berechnen folgende Anfragen?

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



# Programmierung von Endrekursion

- Umsetzung von Rekursion in Endrekursion
    - Verwendung von Akkumulatoren (accumulators)
    - Viele Prolog-Compiler übersetzen Endrekursion in Iteration
    - Rekursiver Aufruf als letzter Befehl
  - Prädikat `listlen` ohne Accumulator
    - `listlen([],0).`
    - `listlen([H|T],N) :- listlen(T,N1), N is N1+1.`
  - Aufrufhierarchie
    - `listlen([a,b,c], N)`
      - `:- listlen([b,c], N1), N is N1+1`
      - `:- listlen([c], N2), N1 is N2+1`
      - `:- listlen([], N3), N2 is N3+1`
-   
=0

# Programmierung mit Akkumulatoren

- Prädikat `listlen` mit Accumulator

- `listlen(L,N) :- lenacc(L,0,N).`  
`lenacc([],A,A).`  
`lenacc([H|T],A,N) :- A1 is A+1, lenacc(T,A1,N).`

- Aufrufe mit Accumulator

- `lenacc([a,b,c,d,e], 0, N)`  
`lenacc([b,c,d,e], 1, N)`  
`lenacc([c,d,e], 2, N)`  
`lenacc([d,e], 3, N)`  
`lenacc([e], 4, N)`  
`lenacc([], 5, N)`

# Cut

- Ziel
  - Doppelte Ausgaben verhindern
  - Unnötiges Durchsuchen verhindern
  - Fehlerhafte Ausgaben verhindern
- Prädikat Cut
  - Backtracking verhindern durch Cut
  - Syntax:  $p :- q1, !, q2.$
  - Wirkung: kein Zurücksetzen auf Prädikate vor dem Cut
- Beispiel
  - $\text{max}(X, Y, Y) :- X \leq Y, !.$
  - $\text{max}(X, Y, X) :- X > Y.$

# Verwendung von Cut

- Cut zur Effizienzsteigerung
  - Vermeidung, dass die zweite Regel benutzt wird, um folgendes herzuleiten: `append(X, [a,b,c,d], Y).`
  - `append([], L, L) :- !.`
  - `append([H|T1], L, [H|T2]) :- append(T1, L, T2).`
- Häufige Unterscheidung
  - Grüner Cut: kann entfernt werden, ohne dass sich die Bedeutung des Programms ändert
  - Roter Cut: Entfernen verändert Bedeutung des Programms

# Verwendung von Cut

- Behandlung von Ausnahmen zu Regeln
  - Alle Vögel können fliegen
  - Strausse und Pinguine sind Vögel, können aber nicht fliegen
- Beschreibung in Prolog
  - `vogel(X) :- strauss(X).`
  - `vogel(X) :- pinguin(X).`
  - `kannfliegen(X) :- strauss(X),!,fail.`
  - `kannfliegen(X) :- pinguin(X),!,fail.`
  - `kannfliegen(X) :- vogel(X).`