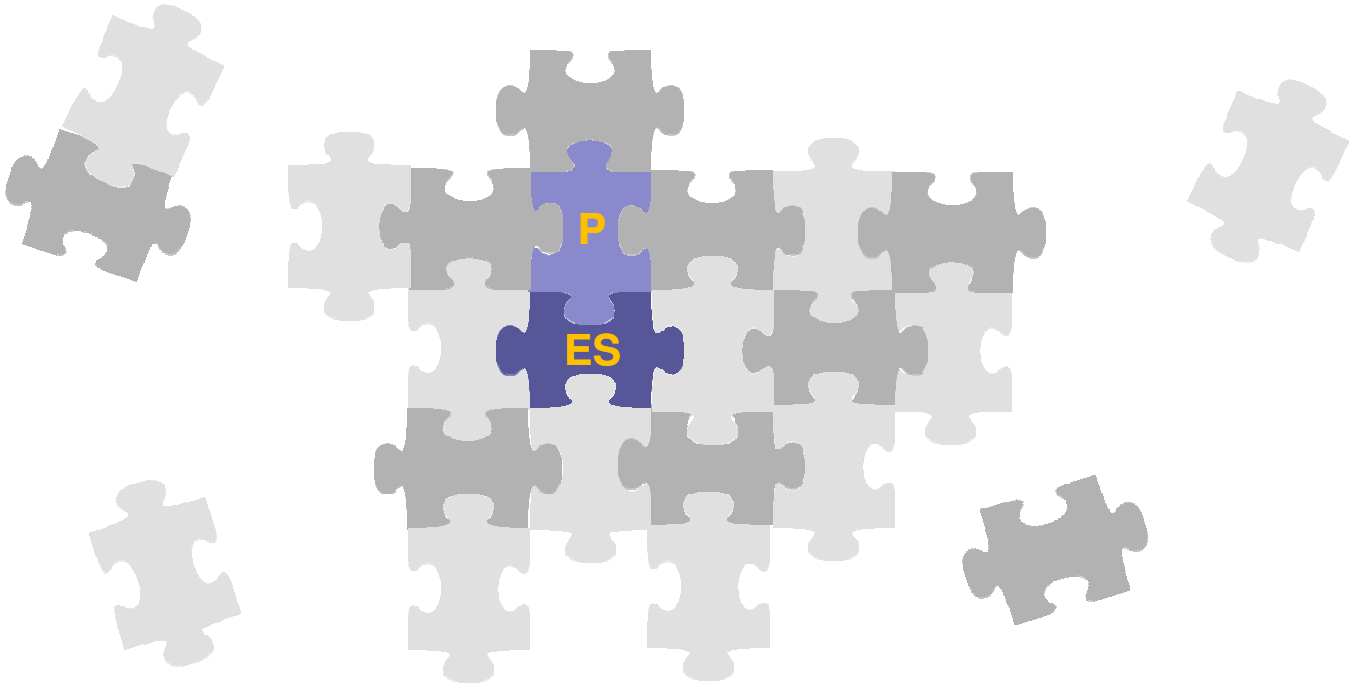
# Vorlesung: „*Künstliche Intelligenz*“

## - Expertensysteme -



# Inhaltliche Planung für die Vorlesung

1) Definition und Geschichte der KI, PROLOG
2) **Expertensysteme**
3) Probabilistisches und Logisches Schließen, Resolution
4) Spieltheorie, Suchen und Planen
5) Spieleprogrammierung
6) General Game Playing
7) Reinforcement Learning und Spieleprogrammierung
8) Mustererkennung
9) Neuronale Netze
10) Optimierungen (genetische und evolutionäre Algorithmen)
11) Bayes-Netze, Markovmodelle
12) Robotik, Pathfinding

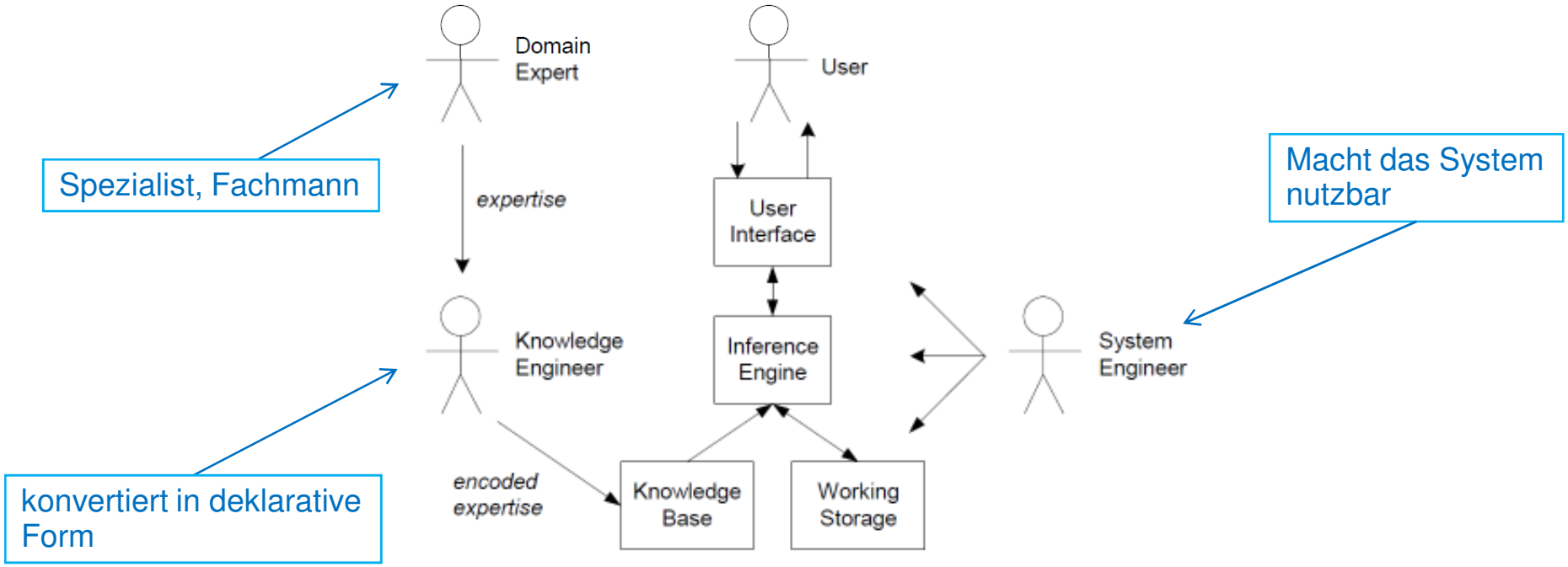
der rote Vorlesungsfaden...

# Wissensbasierte Systeme

Inferenzmechanismus: Trennung von Wissensbasis und Inferenz

Vorteil: Wissensbasis einfach austauschbar, ohne System neu zu programmieren

# Komponenten eines Expertensystems



# Berühmte Expertensysteme

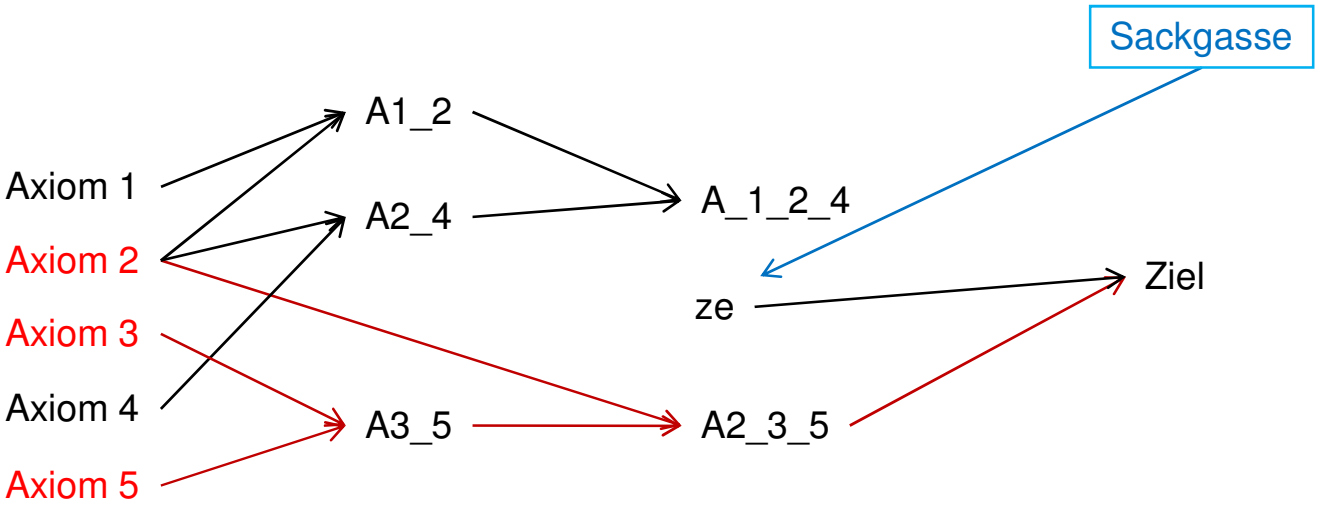
| | |
|---|---|
| DENDRAL | In der Chemie eingesetzt, um unbekannte organische Moleküle zu identifizieren<br>1965 an der Stanford-Universität entwickelt (LISP) |
| MYCIN | Diagnosesystem mit Einsatz in der Medizin<br>1972 an der Stanford-Universität entwickelt (LISP) |

# Merkmale eines Expertensystems

- Ziel- oder datenorientierte Suche
- Unsicherheit
- Datenrepräsentation
- Benutzeroberfläche
- Entscheidungskette zur Lösung

# Ziel- vs. datenorientierte Suche I

Zielorientiert entspricht einer rückwärtsgerichteten Suche (backward chaining)



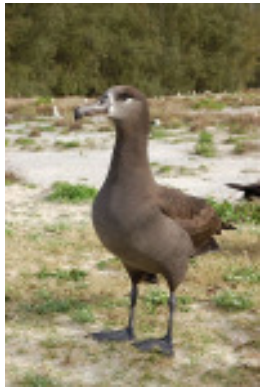

kleine Zielmenge und große Datenmenge

Mathematik, Axiome, geometrische Theorembeweiser

# Ziel- vs. datenorientierte Suche II

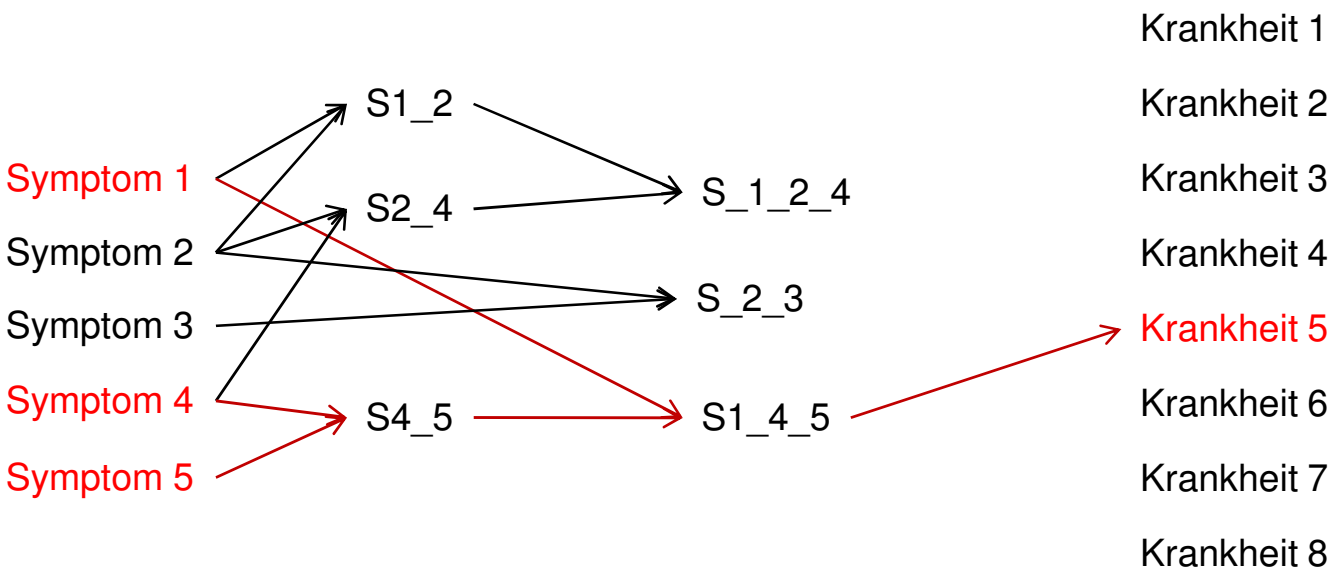
Beispiel:

```
IF
   family is albatross and
   color is white
THEN
   bird is laysan albatross.

IF
   family is albatross and
   color is dark
THEN
   bird is black footed albatross.

IF
   order is tubenose and
   size large and
   wings long narrow
THEN
   family is albatross.
```



# Ziel- vs. datenorientierte Suche III

Datenorientiert entspricht einer vorwärtsgerichteten Suche (forward chaining)




kleine Datenmenge und große Zielmenge, Ziele oft unbekannt

Symptome und Krankheiten

# Ziel- vs. datenorientierte Suche IV

Beispiel:

```
IF
   unplaced tv and
   couch on wall(X) and
   wall(Y) opposite wall(X)
THEN
   place tv on wall(Y).
```

Ein datenorientiertes System muss mit konkreten Daten gefüttert werden. Im Gegensatz dazu sucht sich das zielorientierte System die Daten aus, die es benötigt.

# Datenrepräsentation

## Attribute-Value Pairs

```
color-white
size-large
```

## Object Attribute-Value Triples

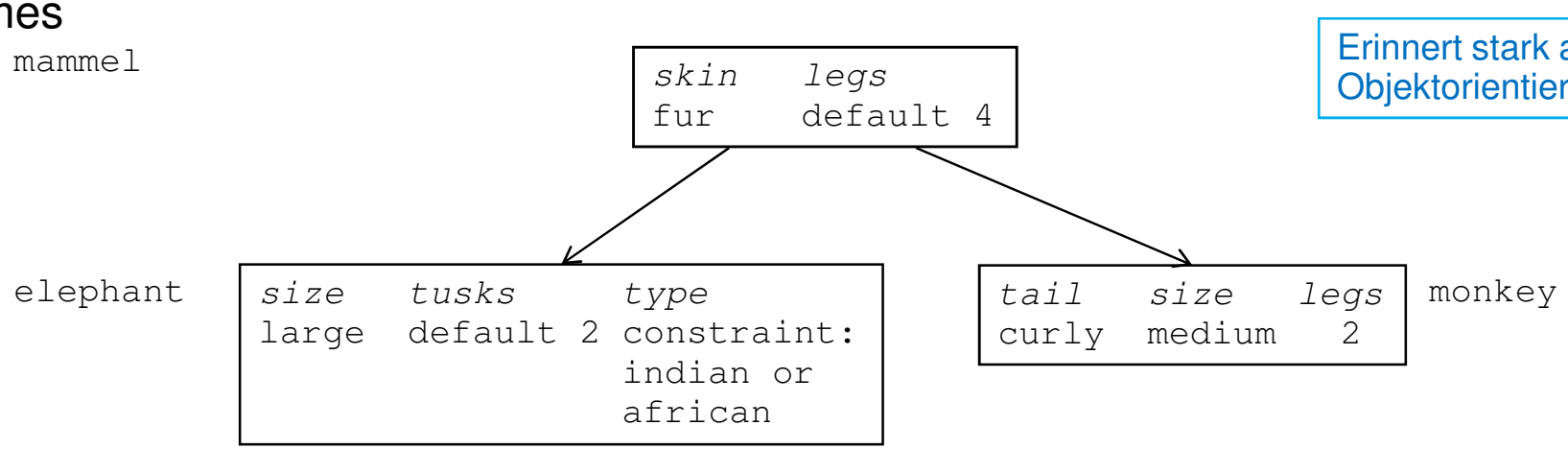
```
arm_chair-width-3
straight_chair-width-2
```

## Records

chairs

| *object* | *width* | *color* | *type* |
|---|---|---|---|
| chair#1 | 3 | orange | easy |
| chair#2 | 2 | brown | straight |

## Frames




Erinnert stark an Objektorientierung

# Entscheidungskette (Explanations)

Das System teilt die Fakten und Regeln mit, die es dazu geführt hatten eine spezielle Lösung zu ermitteln.

Manchmal sind diese aber nicht besonders nützlich, da ein Expertensystem über empirisches Wissen verfügt und kein tiefes Problemverständnis mit sich bringt, sondern lediglich logisch schließt.

# Warum Expertensysteme mit PROLOG?

## Gründe für PROLOG

- Regelbasiertes Programmieren

  Wissen und Regeln können in deklarativer Form besser ausgedrückt werden als in prezeduraler Form
- Built-in-pattern matching
- Backtracking

## Es gibt auch Argumente für konventionelle Sprachen, wie beispielsweise C

- Portability
- Performance
- Developer experience

## Aktuelle PROLOG-Systeme verringern die Vorteile von C.

# Inference Engine von PROLOG

- built-in backward chaining
- es gibt keine Unsicherheit
- adäquat für viele Expertensystem-Applikationen

# Expertensystem: The Bird Identification System (BID)

Die **Expertise** dieses Systems ist eine kleine Untermenge der *Vögel Nordamerikas* von Robbins, Bruum, Zim und Singer.

Typische Form für Regeln, die von Experten gegeben sind:

```
IF
   first premise, and
   second premise, and
   ...
THEN
   conclusion
```

in PROLOG:

```
conclusion :- first_premise,
              second_premise,
              ...
```

# BID-Expertensystem: Einfache Regeln und Fakten

Fangen wir mit den fundamentalen Regeln an:

```
IF
   family is albatross and
   color is white
THEN
   bird is laysan_albatross
```

PROLOG:

```
bird(laysan_albatross) :-
   family(albatross),
   color(white).
```

```
bird(black_footed_albatross) :-
   family(albatross),
   color(dark).

bird(whistling_swan) :-
   family(swan),
   void(muffled_musical_whistle).

bird(trumpeter_swan) :-
   family(swan),
   voice(loud_trumpeting).
```

# BID-Expertensystem: Daten ins Programm und Anfragen

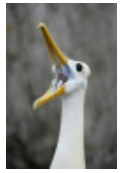
Um eine erfolgreiche Anfrage zu starten, müssen wir zuvor die Fakten des gesuchten Vogels speichern:

```
family(albatross).
color(dark).
```

Bei der Systemanfrage können wir den Vogel jetzt identifizieren:

```
?- bird(X).
X = black_footed_albatross
```

Zu diesem Zeitpunkt haben wir ein komplett lauffähiges Expertensystem, bei dem der PROLOG-Interpreter die Benutzeroberfläche stellt und die Daten direkt im Programm gespeichert sind.



# BID-Expertensystem: Hierarchische Verhältnisse

Jetzt wollen wir natürliche Hierarchien im Vogel-Klassifikationssystem ausdrücken.

```
order(tubenose) :-
   nostrils(external_tubular),
   live(at_sea),
   bill(hooked).

order(waterfowl) :-
   feet(webbed),
   bill(flat).

family(albatross) :-
   order(tubenose),
   size(large),
   wings(long_narrow).

family(swan) :-
   order(waterfowl),
   neck(long),
   color(white),
   flight(ponderous).
```

Ordnung

Familie

Art

War vorher ein einfacher Fakt, jetzt ist es eine Regel.

Die Fakten lassen sich jetzt durch primitivere Daten ausdrücken.

# BID-Expertensystem: Anfrage an das System

Jetzt müssen wir immernoch für eine erfolgreiche Anfrage die relevanten Daten im Programm vorgeben:

```
nostrils(external_tubular).
live(at_sea).
bill(hooked).
size(large).
wings(long_narrow).
color(dark).
```

Wir erhalten für die gleiche Anfrage das gleiche Ergebnis.

```
?- bird(X).
X = black_footed_albatross
```

Das gleiche könnten wir in jeder anderen Programmiersprache realisieren. Expertensysteme zeigen ihre Stärke aber, wenn die Hierarchie nicht klar und die Organisation der Informationen mehr kaotisch ist.

# BID-Expertensystem: Gänse kommen dazu

Die Schneegans (Canada goose) verbringt den Sommer in Kanada und den Winter in den Vereinigten Staaten. Es ist also wichtig, welche Saison vorliegt und wo sie gesehen wurde.

```
bird(canada_goose) :-
   family(goose),
   season(winter),
   country(united_states),
   head(black),
   cheek(white).

bird(canada_goose) :-
   family(goose),
   season(summer),
   country(canada),
   head(black),
   cheek(white).
```

```
country(united_states) :-
   region(mid_west),
   ...

country(canada) :-
   province(ontario),
   ...

region(south_east) :-
   state(X),
   member(X, [florida,
              mississippi,
              ...]).
...
```

Die Prädikate können sich weiter in komplexere Hierarchien verzweigen.

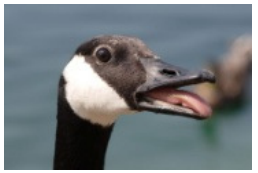
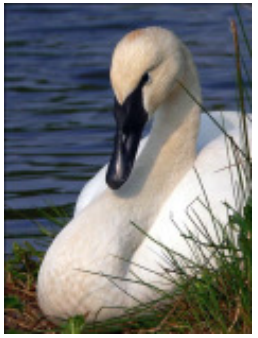
# BID-Expertensystem: Geschlechterspezifische Daten

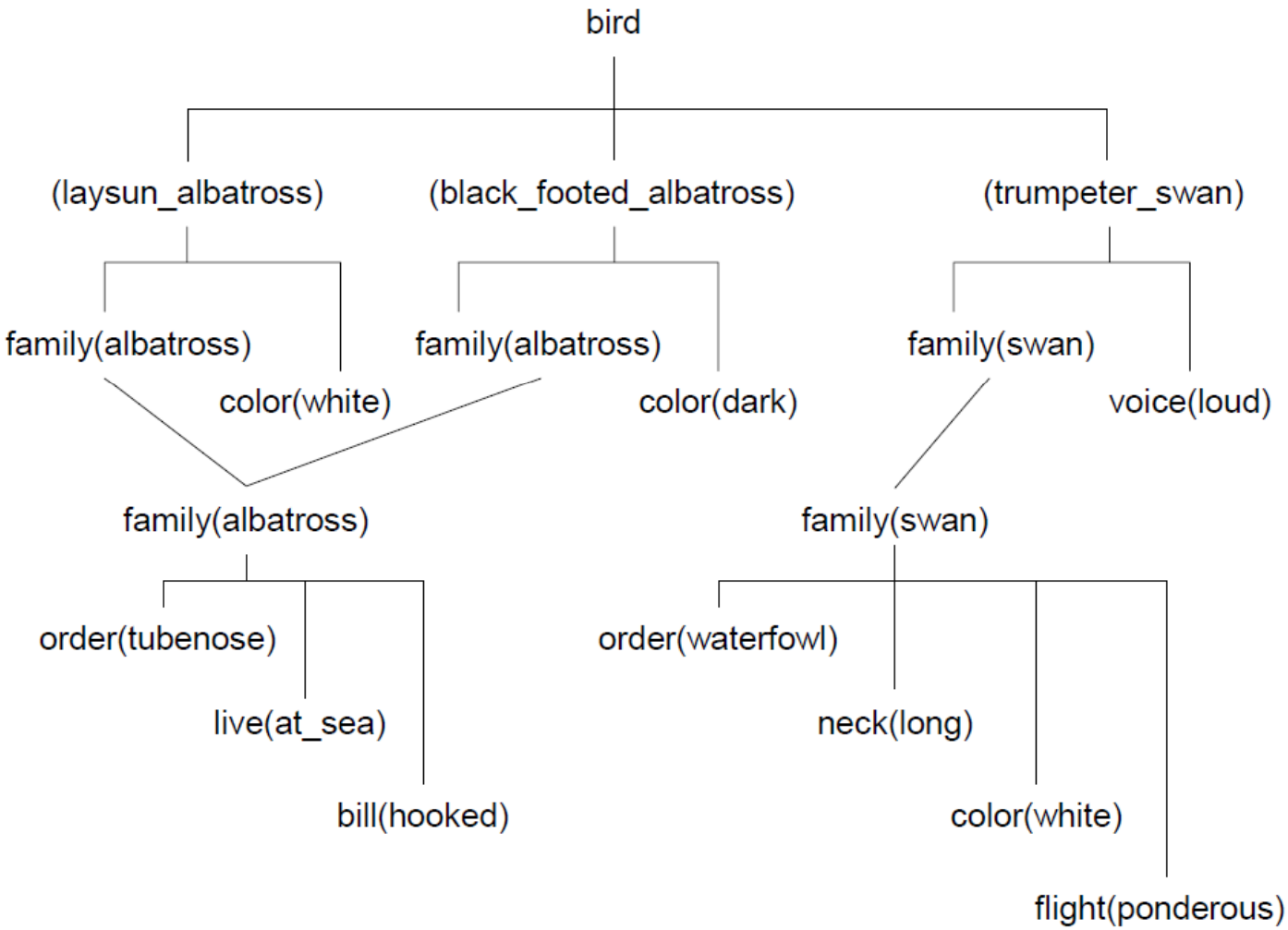
## Unterscheidung zweier Geschlechter

```
bird(mallard) :-
   family(duck),
   voice(quack),
   head(green).

bird(mallard) :-
   family(duck),
   voice(quack),
   color(mottled_brown).
```

# BID-Expertensystem: Übersicht einer Regeluntermenge





# BID-Expertensystem: Benutzerschnittstelle

Bisher wurden die Daten in das Programm eingegeben und die Anfragen waren sehr einfach.

Das System soll Anfragen an den Benutzer stellen, wenn diese für die Entscheidung wichtig sind. Das Prädikat **ask** wird vom Benutzer für ein gegebenes Attribute-Value-Pair die Antwort wahr oder falsch einlesen.

```
eats(X)  :- ask(eats, X).
feet(X)  :- ask(feet, X).
wings(X) :- ask(wings, X).
neck(X)  :- ask(neck, X).
color(X) :- ask(color, X).
```

Wenn das System jetzt color(white) entscheiden muss, erfragt es ask(color, white) und falls das positiv (also yes) ist, führt auch color(white) zum Erfolg.

# BID-Expertensystem: Prädikat ask I

Dabei ist das Prädikat read erfüllt, wenn der Benutzer „yes“ eingibt und

```
ask(Attr, Val) :-
   write(Attr:Val),
   write(`?`),
   read(yes).
```

fail sonst.

Bei der Anfrage ergibt sich daraus folgender Dialog:

```
?- bird(X).
nostrils : external_tubular? yes.
live : at_sea? yes.
bill : hooked? yes.
size : large? yes.
wings : long_narrow? yes.
color : white? yes.
X = laysan_albatross
```

# BID-Expertensystem: Prädikat ask II

Bei diesem Ansatz ergibt sich aber ein Problem:

Falls der Benutzer bei der letzten Frage mit „no“ antwortet, würde die Regel bird(laysan_albatross) ein fail liefern und die nächste Regel bird(black_footed_albatross) testen.

```
bird(laysan_albatross) :-
   family(albatross),
   color(white).

bird(black_footed_albatross) :-
   family(albatross),
   color(dark).
...
```

Die gleichen Fragen werden gestellt...

Dabei wird aber die erste Regel family(albatross) wieder erfragt...

Wir wollen also die Antworten speichern.

# BID-Expertensystem: Prädikat ask III

Wir erzeugen ein neues Prädikat known/3. Die Zahl 3 verrät uns dabei, dass es sich um ein Prädikat mit 3 Argumenten handelt.

```
ask(A, V) :-
   known(yes, A, V),
   !.

ask(A, V) :-
   known(_, A, V),
   !,
   fail.

ask(A, V) :-
   write(A:V),
   write(`? : `),
   read(Y),
   asserta(known(Y, A, V)),
   Y==yes.
```

Wir gehen bei dieser Lösung aber davon aus, dass ein Attribute-Value-Pair entweder wahr oder falsch ist.

# BID-Expertensystem: Prädikat ask IV

Wir erweitern unser System

```
multivalued(voice).
multivalued(feed).
```

Jetzt können wir eine weitere Klausel im ask-Prädikat unterbringen:

```
ask(A, V) :-
   not(multivalued(A),
   known(yes, A, V2),
   V \== V2,
   !,
   fail.
```

```
ask(A, V) :-
   known(yes, A, V),
   !.

ask(A, V) :-
   known(_, A, V),
   !,
   fail.

ask(A, V) :-
   write(A:V),
   write(`? : `),
   read(Y),
   asserta(known(Y, A, V)),
   Y==yes.
```

# BID-Expertensystem: Menüs

Das Menü kann erweitert werden, indem es dem Benutzer eine Liste der möglichen Attribut-Werte liefert.

```
size(X)   :- menuask(size, X, [large, plump, medium, small]).
flight(X) :- menuask(fligth, X, [ponderous, agile, flap_glide]).
```

Prädikat menuask:

```
menuask(Attribute,Value,_) :-
   known(yes,Attribute,Value), !.

menuask(Attribute,_,_)     :-
   known(yes,Attribute,_), !, fail.

menuask(A, V, MenuList) :-
   write('What is the value for'), write(A), write('?'), nl,
   write(MenuList), nl,
   read(X),
   check_val(X, A, V, MenuList),
   asserta( known(yes, A, X) ),
   X == V.

check_val(X, A, V, MenuList) :-
   member(X, MenuList), !.

check_val(X, A, V, MenuList) :-
   write(X), write(' is not a legal value, try again.'), nl,
   menuask(A, V, MenuList).
```

# BID-Expertensystem: Einfache Shell

Ein Highlevel-Prädikat wird eingeführt, um mit minimalem Aufwand die Wissensbasis austauschen zu können:

```
top_goal(X) :- bird(X).
```

Mit dem folgenden Prädikat läßt sich das System „aufräumen“:

```
solve :-
   abolish(known, 3),
   define(known, 3),
   top_goal(X),
   write(`The answer is `), write(X), nl.

solve :-
   write(`No answer found.`), nl.
```

abolish entfernt alle bekannten Einträge von known/3. Bei der ersten Referenzierung eines Prädikats muss die Signatur bekannt sein, daher define(known, 3).

# BID-Expertensystem: Wissensbasis und Identifikationssystem I

Der Code wird jetzt in zwei Dateien gelagert:

```
...

solve :-
   abolish(known,3),
   prove(top_goal(X),[]),
   write('The answer is '),write(X),nl.

solve :- write('No answer found.'),nl.

ask(Attribute,Value,_) :-
   known(yes,Attribute,Value), !.

ask(Attribute,Value,_) :-
   known(_,Attribute,Value), !, fail.

ask(Attribute,_,_)     :-
   not multivalued(Attribute),
   known(yes,Attribute,_), !, fail.

ask(A,V,Hist)          :-
   write(A : V), write('? (yes or no) '),
   get_user(Y,Hist),
   asserta(known(Y,A,V)),
   Y = yes.

...
```

```
...

top_goal(X) :- bird(X).

order(tubenose)     :-
   nostrils(external_tubular),
   live(at_sea),
   bill(hooked).

order(waterfowl)    :-
   feet(webbed),
   bill(flat).

order(falconiforms) :-
   eats(meat),
   feet(curved_talons),
   bill(sharp_hooked).

order(passerformes) :-
   feet(one_long_backward_toe).

family(albatross) :-
   order(tubenose),
   size(large),
   wings(long_narrow).

family(swan) :-
   order(waterfowl),
   neck(long),

...
```

Prädikate der Shell

Wissensbasis

# BID-Expertensystem: Wissensbasis und Identifikationssystem II

Die Kombination beider sieht jetzt wie folgt aus:

```
?- consult(native).
yes

?- consult(`birds.kb`).
yes

?- solve.
nostrils : external_tubular?
...
```

Das ganze läßt sich noch erweitern... Nachzulesen in [5]

# Backward Chaining mit Unsicherheit

Das Bird Identification System war ein gutes Beispiel für die Erstellung eines Expertensystems. Dabei hatten wir aber die Annahme, dass alle Informationen entweder wahr oder falsch sein können.

Beispielsweise könnte das Gefieder grau sein und daher nicht eindeutig in schwarz oder weiss klassifiziert werden.

Wir wollen Unsicherheit, also Attribute mit Wahrscheinlichkeiten, modellieren.

# Backward Chaining mit Unsicherheit

Die am häufigsten verwendete Methode mit Unsicherheit zu arbeiten:

Der Bestimmtheits- oder Sicherheitsfaktor (certainty factor) wird an jede Information angehaftet und bei Kombinationen oder Schlüssen durch die Inferenzengine automatisch aktualisiert.

Certainty factor (cf) ist dabei im folgenden Integerwert zwischen -100 (definitiv false) und 100 (definitiv true).

# Backward Chaining mit Unsicherheit (Clam)

```
goal problem.

rule 1
if not turn_over and
battery_bad
then problem is battery.

rule 2
if lights_weak
then battery_bad cf 50.

rule 3
if radio_weak
then battery_bad cf 50.

rule 4
if turn_over and
smell_gas
then problem is flooded cf 80.

rule 5
if turn_over and
gas_gauge is empty
then problem is out_of_gas cf 90.

rule 6
if turn_over and
gas_gauge is low
then problem is out_of_gas cf 30.
```

```
ask turn_over
menu (yes no)
prompt 'Does the engine turn over?'.

ask lights_weak
menu (yes no)
prompt 'Are the lights weak?'.

ask radio_weak
menu (yes no)
prompt 'Is the radio weak?'.

ask smell_gas
menu (yes no)
prompt 'Do you smell gas?'.

ask gas_gauge
menu (empty low full)
prompt 'What does the gas gauge say?'.
```

```
:consult
Does the engine turn over? : yes
Do you smell gas? : yes
What does the gas gauge say?
empty, low, full : empty
problem-out_of_gas-cf-90
problem-flooded-cf-80
done with problem
```

Im Gegensatz zu PROLOG werden alle Lösungen mitgeteilt (cf)

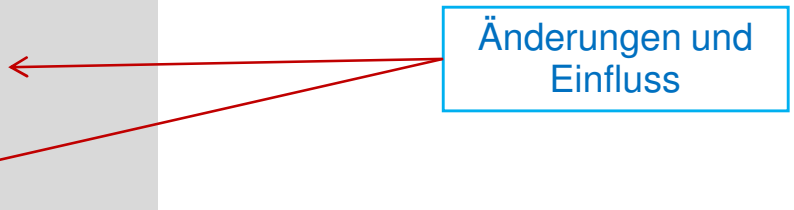
# Backward Chaining mit Unsicherheit (Clam)

Beispiel zuvor:

```
:consult
Does the engine turn over? : yes
Do you smell gas? : yes
What does the gas gauge say?
empty, low, full : empty
problem-out_of_gas-cf-90
problem-flooded-cf-80
done with problem
```

So können wir cf-Parameter mit eingeben:

```
:consult
Does the engine turn over? : yes
Do you smell gas? : yes cf 50
What does the gas gauge say?
empty, low, full : empty
problem-out_of_gas-cf-90
problem-flooded-cf-40
done with problem
```



# Certainty Factors in MYCIN

Wir haben die zwei Möglichkeiten CFs in das System zu bekommen: Regeln, Input

Die CFs einer Folgerung basiert auf den CFs der Prämissen.

Beispiel: Wenn die Folgerung eine CF von 80 hat, und die Prämisse ist mit CF = 100 bekannt, dann wird die Folgerung mit CF = 80 gespeichert.

```
turn_over cf 100
smell_gas cf 100

rule 4
if turn_over and
smell_gas
then problem is flooded cf 80

problem flooded cf 80
```

# Certainty Factors in MYCIN

Kombination von CF der Prämisse und CF der Folgerung

```
CF = RuleCF * PremiseCF / 100
```

das ergibt für unser Beispiel

```
turn_over cf 80
smell_gas cf 50

rule 4
if turn_over and
smell_gas
then problem is flooded cf 40

problem flooded cf 40
```

# Certainty Factors in MYCIN

Wir wollen verhindern, dass sich alle Regeln angesprochen fühlen und feuern.

Schwellenwert für PremiseCF wird benötigt (20 dient dabei als untere Schranke):

```
turn_over cf 80
smell_gas cf 15
```

Regel 4 würde jetzt also nicht feuern, da eine der Prämissen zu niedrig ist.

Kombination verschiedener CFs

```
CF(X, Y) = X+Y(100-X)/100.          % wenn X und Y > 0
CF(X, Y) = X+Y/1 - min (|X|,|Y|). % wenn entweder X oder Y < 0
CF(X, Y) = -CF(-X, -Y).             % wenn X und Y < 0
```

Beispiel:

```
turn_over cf 80
smell_gas cf 15
```

# Certainty Factors in MYCIN

Beispiel:

```
rule 2
if lights_weak
then battery_bad cf 50.

rule 3
if radio_weak
then battery_bad cf 50.
```

angenommen, es gelten die folgenden Parameter

```
lights_weak cf 100
radio_weak cf 100
```

dann folgt aus Regel 2

```
battery_bad cf 50
```

und aus Regel 3

```
battery_bad cf 75
```

Das zeigt, dass Clam eine neue Inference-Engine benötigt. Immer wenn wir CF<100 haben wollen wir alle kombinieren und nicht nur immer eine Regel verwenden.

# Eigene Inferenz-Engine - Notation

Wir führen folgende Notation ein:

```
rule(Name, LHS, RHS).    % mit LHS => RHS oder RHS :- LHS
                         % LHS Prämissen
                         % RHS Schlussfolgerung
```

```
rhs(Goal, CF)
lhs(GoalList)
```

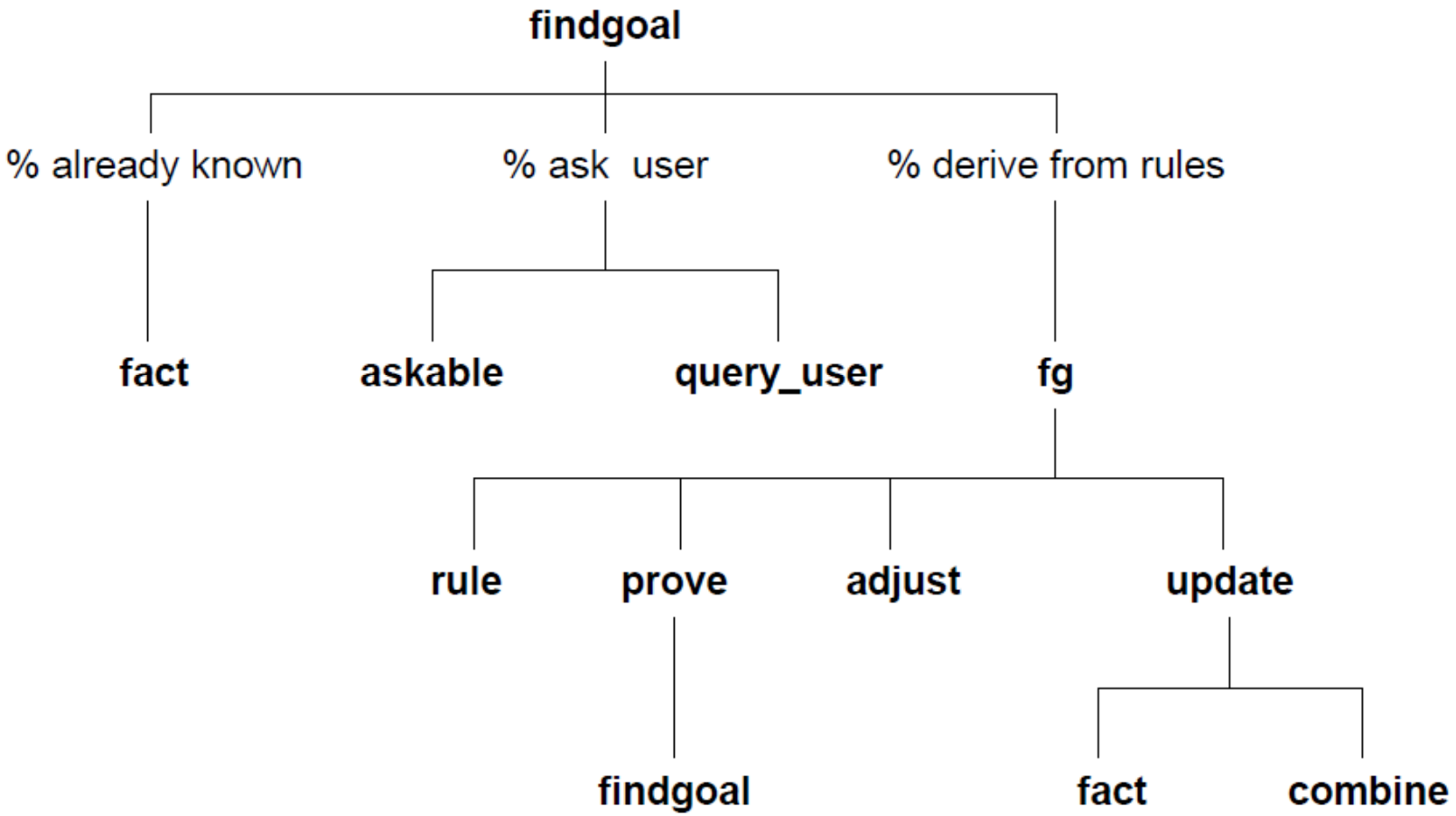
Attribute-Value-Pairs:

```
av(Attribute, Value)

rule(Name,
     lhs([av(A1, V1), av(A2, V2), ...]),
     rhs(av(Attr, Val), CF)).
```

Beispiel mit Regel 5:

```
rule(5, lhs([av(turns_over, yes, av(gas_gauge, empty)]),
     rhs(av(problem, flooded), 80)).
```

# Eigene Inferenz-Engine - Übersicht



# Eigene Inferenz-Engine - fact

Bekannte Fakten werden einfach gespeichert:

```
fact(av(A, V), CF).
```

Frage an das System:

```
?- findgoal(av(problem, X), CF).
```

Drei Fälle können bei der Bearbeitung auftreten:

- Attribute-Value-Pair ist bekannt
- Es gibt Regeln aus denen sich das Attribute-Value-Pair ableiten läßt
- Wir müssen den Benutzer fragen

# Eigene Inferenz-Engine - fact

## Definition von findgoal

```
findgoal( av(Attr, Val), CF) :-
   fact( av(Attr, Val), CF), !.

findgoal(av(Attr, Val), CF) :-
   not fact(av(Attr, _), _),
   askable(Attr, Prompt),
   query_user(Attr, Prompt),
   !,
   findgoal(av(Attr, Val), CF).

findgoal(Goal, CurCF) :-
   fg(Goal, CurCF).

fg(Goal, CurCF) :-
   rule(N, lhs(IfList), rhs(Goal, CF)),
   prove(IfList, Tally),
   adjust(CF, Tally, NewCF),
   update(Goal, NewCF, CurCF),
   CurCF == 100, !.
   fg(Goal, CF) :- fact(Goal, CF).
```

# Literatur und Abbildungsquellen

[1] Ertel, W.: „*Grundkurs Künstliche Intelligenz*“, Vieweg Verlag 2007

[2] Braitenberg V.: „*Vehicles – Experiments in Synthetic Psychology*“, MIT Press, 1984

[3] Bratko, I.: „*PROLOG Programming for Artificial Intelligence*“, 3.Auflage, Pearson Verlag 2001

[4] Copeland J.: „*Artificial Intelligence: A Philosophical Introduction*“, Oxford UK and Cambridge, 1993

[5] Merrit D.: „*Building Expert Systems in Prolog*“, Openbook, Amzi!, 2000