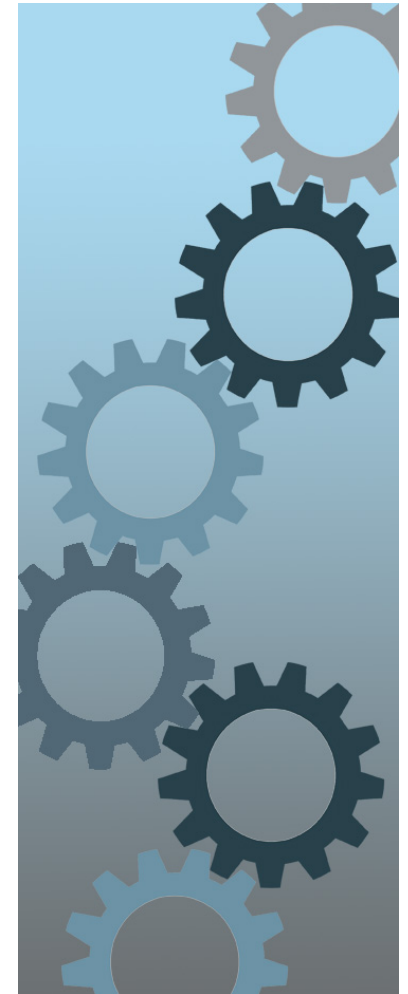


Programming Concepts & Paradigms

Prolog 1

Prof. Dr. Ruedi Arnold

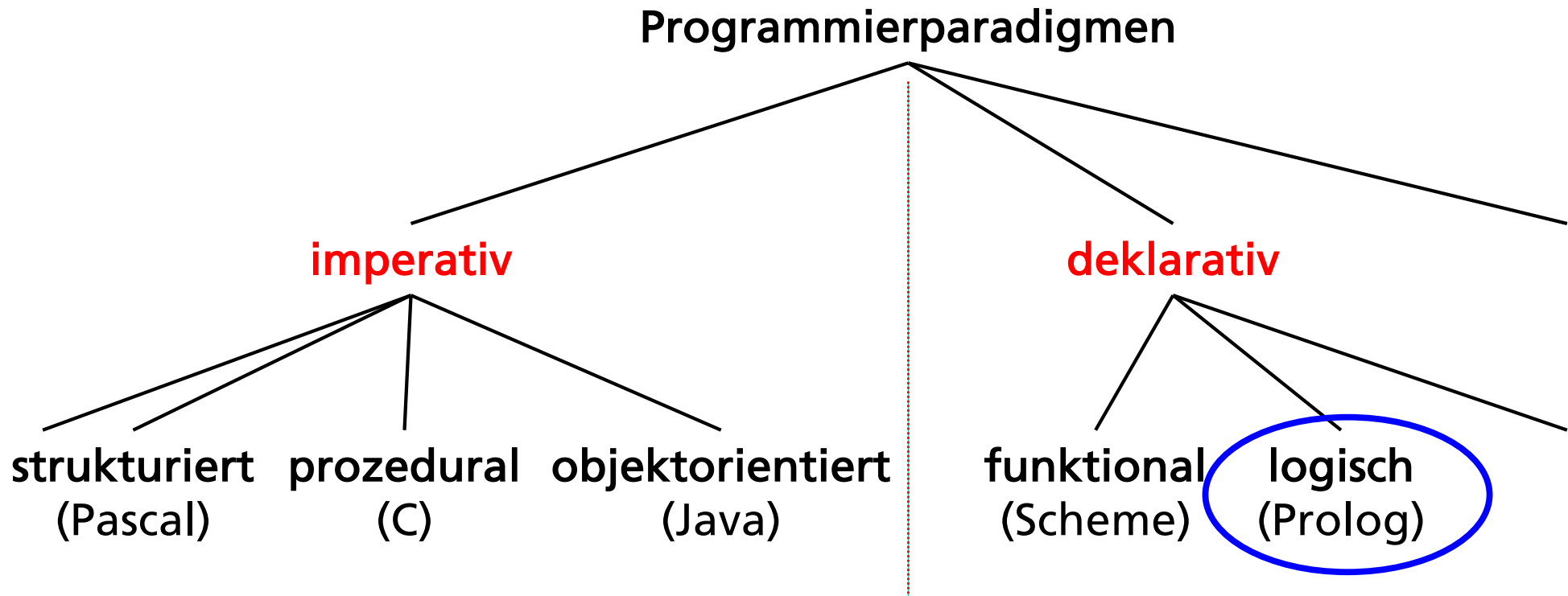
ruedi.arnold@hslu.ch



Übersicht Prolog 1

- Prolog-Einstieg
- Wissensdatenbank: Fakten & Regeln
- Anfragen an das System & Antworten vom System
- Syntax: Terme, Atome, Variablen, ...
- Prädikate (eingebaute, transitive Hülle, Stelligkeit, ...)
- Exkurs: Unendliche Terme & der Occurs Check
- Beweissuche: Backtracking & Matching
- Deklarative und prozedurale Bedeutung von Programmen

Wiederholung: Paradigmen & typische Sprache



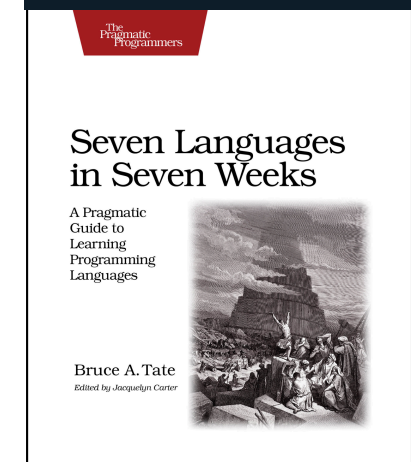
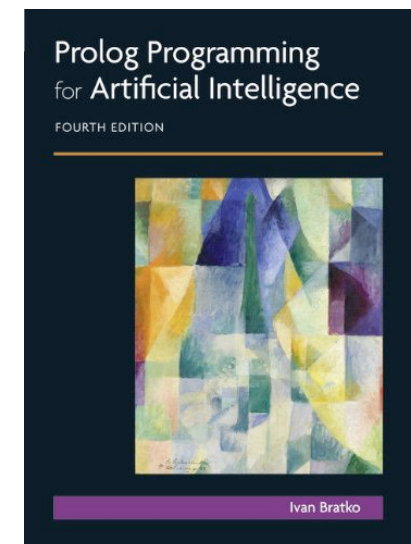
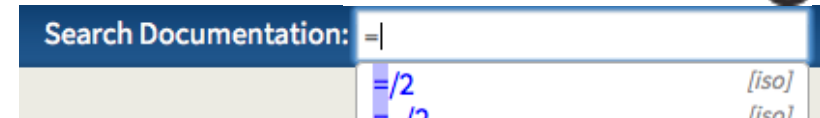
→ Die nächsten drei Wochen im Fokus: **deklarativ-logische Programmierung** am Beispiel Prolog



SWI Prolog

Quellen / Referenzen zum Prolog-Teil

- SWI-Prolog: <http://www.swi-prolog.org>
 - „unsere“ Prolog-Implementierung
 - Web-Such-Funktion: Nutzen!
 - Ich werde öfters SWI-Doku von Prädikaten zeigen...
- Buch „Prolog Programming for Artificial Intelligence (4th Edition)“ von Ivan Bratko
 - Sehr gute Prolog-Einführung
 - Beispiele teilweise daraus entnommen
- Buch „Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages“ von Bruce A. Tate
 - Guter, sehr kompakter Prolog-“Crash Course“



Hinweise zum Prolog-Teil: selber programmieren!

- Programmieren lernt man, indem man es tut! 😊
- Es hat viele Prolog-Beispiele auf den Folien:
versuchen sie, diese nachzuvollziehen
 - ggf. zuhause oder in Übungsstunden selber laufen lassen
 - Code verfügbar im GitLab-Repo: 1 Datei pro Foliensatz
 - <https://gitlab.enterpriselab.ch/PCP/PCP-public-Code>
 - Namensschema: PrologX.pl (X = Nr. Foliensatz, d.h. 1-6)
- Lösen sie die Übungen!
 - ggf. mit Hilfe von Kommilitonen, Assistierenden oder Dozierenden...
 - Selber durchdenken = grösster Lerneffekt 😊

PROLOG = PROgrammation en LOGique

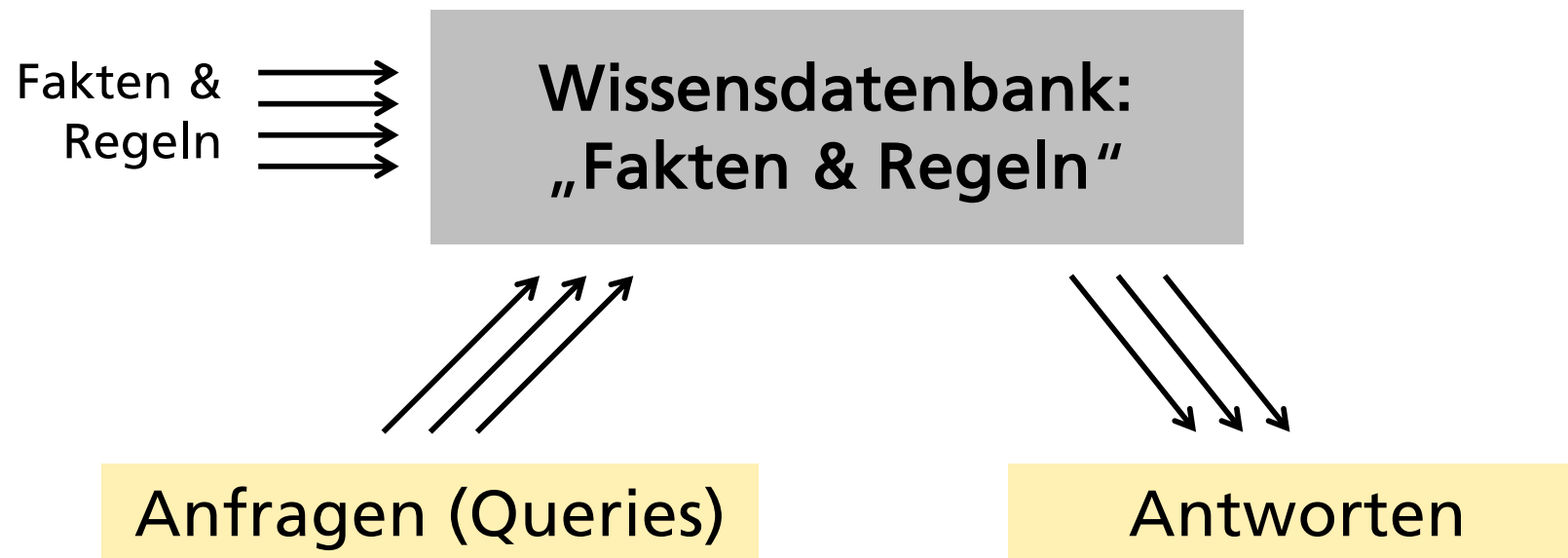
- **Prolog: Deklarative Programmierung**
 - Beschreibt mit „Logik“ (Relationen) **was** für ein Resultat wir wollen (und nicht **wie** das berechnet werden soll!)
- **Wichtigste Mechanismen (siehe später)**
 - Matching
 - Automatisches Backtracking
- **Entstand in den frühen 1970er Jahren**
 - Entwickler: Alain Colmerauer mit Philippe Roussel
- **Basierend auf Prädikatenlogik erster Stufe**
 - Notation von Regeln entspricht Horn-Klauseln (siehe später)



<http://alain.colmerauer.free.fr/aicol/ArchivesPublications/photos/australie.jpg>

„Big Picture“: Funktionsweise von Prolog

- Wissensdatenbank (Knowledge Base)
 - Bestehend aus Fakten & Regeln
 - Kann abgefragt werden durch Anfragen (Queries)



Einstiegsbeispiel: Fakten in der Wissensdatenbank

- Ein erstes Prolog-Programm, bestehend aus den folgenden drei Beispiel-Fakten in der Wissensdatenbank

```
bigger(elephant, horse) .  
bigger(horse, dog) .  
bigger(horse, sheep) .
```

- Diese drei Fakten (resp. drei Relationen) bedeuten für uns:
 - „ein Elefant ist grösser als ein Pferd“
 - „ein Pferd ist grösser als ein Hund“
 - „ein Pferd ist grösser als ein Schaf“

Hinweis: Einträge in der aktuellen Wissensdatenbank (d.h. **Programme**) werden auf diesen Folien grundsätzlich auf **grauem Hintergrund** dargestellt

Bemerkungen zu bigger(...)

```
bigger(elephant, horse) .
```

- `bigger` ist nicht vordefiniert vom System
 - `bigger(elephant, horse)` hat für Prolog keine Bedeutung, das definiert einfach eine Relation, von der wir die Bedeutung kennen. Hier also z.B. ein Elefant ist grösser als ein Pferd
- Den Namen „bigger“ haben wir festgelegt!
 - Wir hätten auch `groesser(...)` oder `xyz(...)` wählen können
 - Analog zur Wahl von Bezeichnern (Identifiers) in anderen Programmiersprachen (z.B. Java)
 - Grundsatz: Wir verwenden (möglichst) sprechende Namen!
 - Ganz im Sinn von „Clean Code“ 😊

Anfragen an die Wissensdatenbank

- Bsp.-Frage 1: „Ist ein Hund grösser als ein Elefant?“

```
?- bigger(dog, elephant) .
```

– Antwort vom Prolog-System: **false**.

- Bsp.-Frage 2: „Ist ein Elefant grösser als ein Pferd?“

```
?- bigger(elephant, horse) .
```

– Antwort vom Prolog-System: **true**.

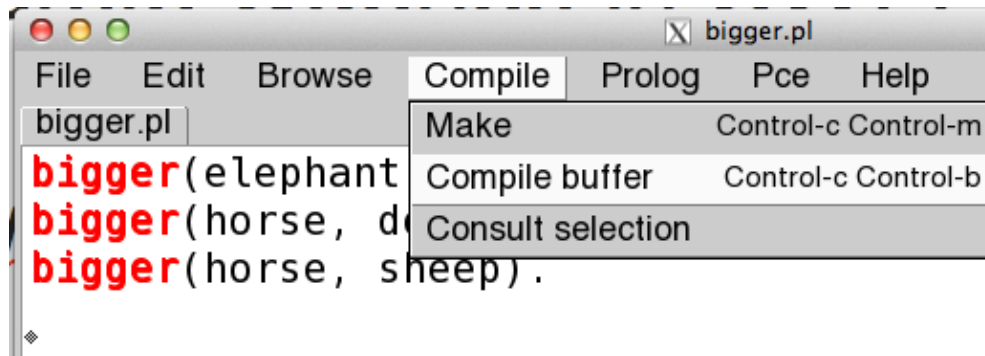
Hinweis: **Anfragen** in der Prolog-Konsole an die Wissensdatenbank sind auf diesen Folien grundsätzlich auf **gelbem Hintergrund** dargestellt

Demo: erste Anfragen in SWI-Prolog



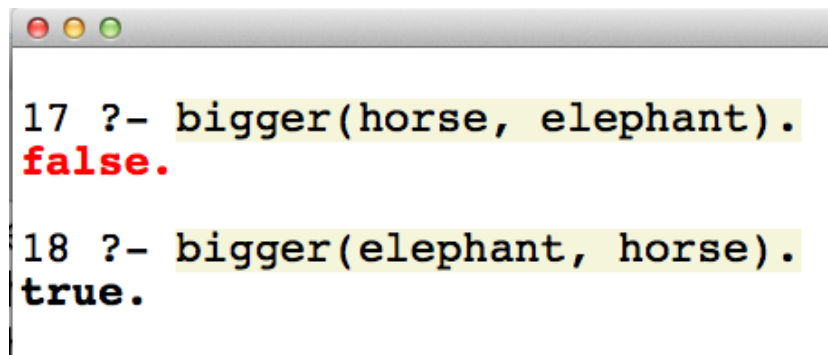
SWI Prolog

- <http://www.swi-prolog.org/>
 - Editor, Inhalt wird kompiliert in die Wissensdatenbank



auf den Folien
grau hinterlegt

- Kommandozeile: Anfrage und Antwort vom System



auf den Folien
gelb hinterlegt

Anfrage mit einer Variablen

■ Bsp.-Anfrage 3: „Ein Pferd ist grösser als wer?“

```
?- bigger(horse, X) .
```

– Antwort vom Prolog-System: **X = dog**

– Weitere Lösung gewünscht?

- Ja: Taste „;“

- Antwort vom Prolog-System: **X = sheep.**

- Nein: Taste „.“

- Abbruch dieser Anfrage

```
23 ?- bigger(horse, X) .  
X = dog  
X = sheep.
```



„.“ bedeutet hier: „keine weiteren Lösungen“

Problem: Elefant nicht grösser als Hund?

■ Bsp.-Frage 4: „Elefant grösser als Hund?“

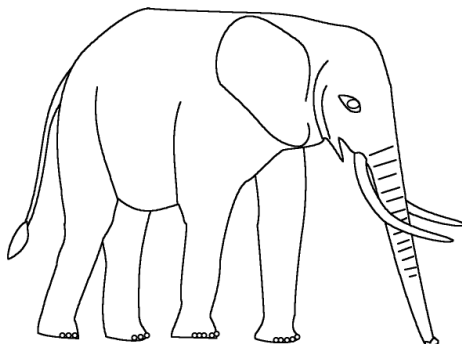
```
?- bigger(elephant, dog).
```

– Antwort vom Prolog-System: **false**.

– Was läuft hier falsch?

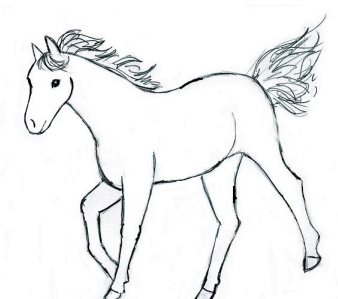
```
bigger(elephant, horse).  
bigger(horse, dog).
```

- Offensichtlich ist unser Prädikat `bigger/2` nicht transitiv... (/2, da `bigger` 2 Argumente hat, siehe später)



<http://www.enchantedlearning.com/bwbig/elephant.png>

>



<http://www.art-made-easy.com/images/horse-drawing5.jpg>

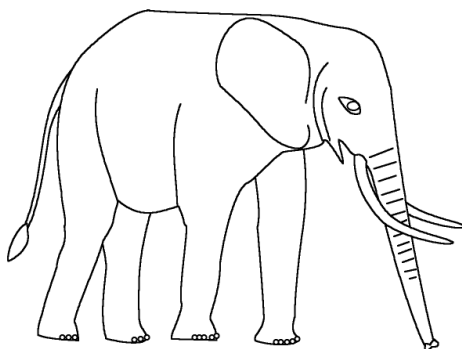
>



<http://www.clipartbest.com/cliparts/aie/xEj/aiexEjei4.jpeg>

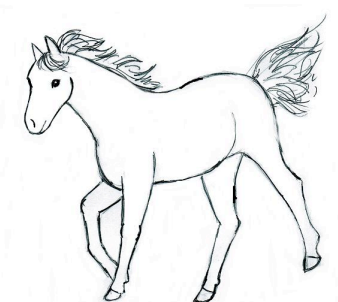
Lösung: transitive Hülle

- Wir hätten gerne die transitive Hülle von `bigger/2`!
 - D.h. ein Prädikat, mit welchem ein Tier A grösser ist als ein anderes Tier B, wenn wir dazwischen mit den früher definierten Fakten über andere Tiere iterieren können
 - Also hier praktisch: Wenn ein Elefant grösser ist als ein Pferd und ein Pferd grösser ist als ein Hund, dann soll auch ein Elefant grösser sein als ein Hund



<http://www.enchantedlearning.com/bwbig/elephant.png>

>



<http://www.art-made-easy.com/images/horse-drawing5.jpg>

>



<http://www.clipartbest.com/cliparts/aie/xEj/aiexEjei4.jpeg>

Neue Regeln für transitive Hülle: is_bigger/2

- Die zwei folgenden Regeln definieren die transitive Hülle von `bigger/2` (via Rekursion)

```
is_bigger(X, Y) :- bigger(X, Y).  
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```



„if“



„and“

- Und dann klappt's auch wie gewünscht

```
?- is_bigger(elephant, dog).
```

– Antwort vom Prolog-System: **true.**

Neue Regel funktioniert auch mit Variabel

- Wer ist grösser als ein Hund?

```
?- is_bigger(X, dog) .  
X = horse ;  
X = elephant ;  
false.
```

- Ein Elefant ist grösser als wer?

```
?- is_bigger(elephant, X) .  
X = horse ;  
X = dog ;  
X = sheep ;  
false.
```


Zwei weitere Beispiel-Anfragen

- Gibt es ein Tier, das kleiner als ein Elefant und grösser als ein Hund ist?

```
?- is_bigger(elephant, X), is_bigger(X, dog).  
X = horse ;  
false.
```

- Gibt es ein Tier, das kleiner als ein Pferd und grösser als ein Hund ist?

```
?- is_bigger(horse, X), is_bigger(X, dog).  
false.
```

Kontrollfragen A

1. Wie antwortet das Prolog-System in unserem Beispiel auf die Anfrage `is_bigger(elephant, fly)` ?
2. Was ist in unserem Beispiel der Unterschied zwischen `bigger(X, Y)` und `is_bigger(X, Y)` ?
3. Definieren sie für das gegebene Beispiel eine Regel für `is_smaller(X, Y)`. (Hinweis: Das geht in einer Zeile)



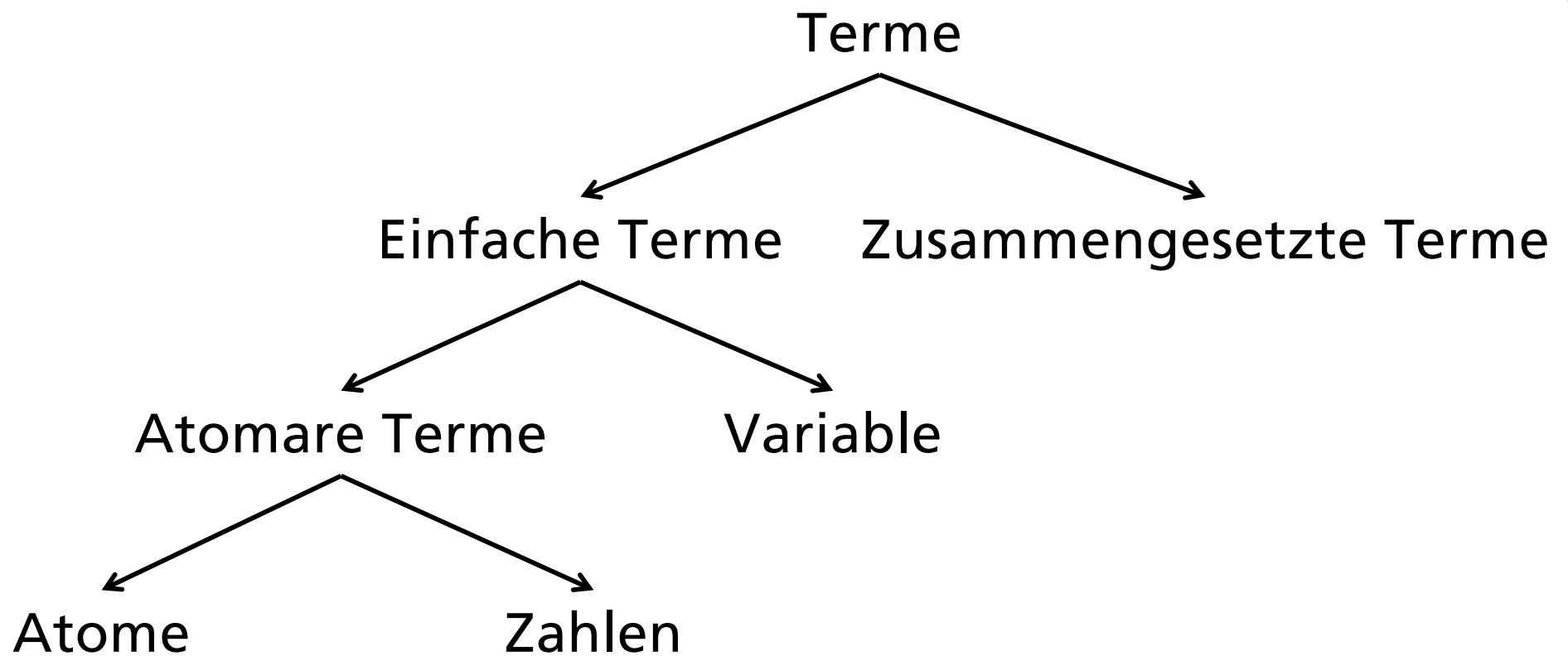
<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Syntax: Terme, Atome, Variablen, ...

Prolog Basissyntax: Terme

- Prolog-Terme sind ausschliesslich:
 - **Zahlen** (numbers)
 - z.B.: 123, 4567.8, -9
 - **Atome** (atoms)
 - z.B.: elephant, 'mein text'
 - **Variablen** (variables)
 - z.B.: X, Elephant oder _
 - **Zusammengesetzte Terme** (compound terms)
 - z.B.: is_bigger(horse, X)

Prolog Terme: Übersicht



Atome und Variablen

- **Atome:** beginnen mit Kleinbuchstaben oder sind eingeschlossen in einfache Anführungszeichen
 - z.B.: elephant, a_bc, 'Hallo mein Text', is_bigger
- **Variablen:** beginnen mit Grossbuchstaben oder einem Unterstrich (underscore)
 - z.B. X, Elephant, _, _elephant

Die anonyme Variable _

- Die Variable _ (Unterstrich) heisst **anonyme Variable**
 - Zweck: Platzhalter, dessen Wert nicht interessiert
- Jedes Auftreten von _ repräsentiert eine neue Variable
- Instanziierungen von _ werden nicht ausgegeben
 - Beispiel:

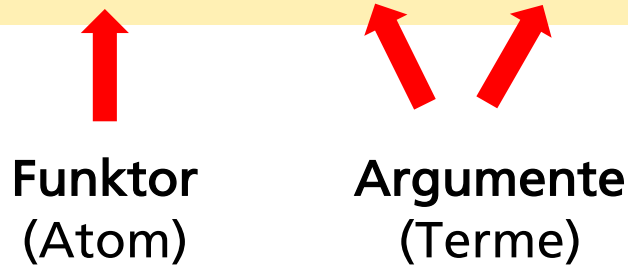
```
?- is_bigger(_, horse) .  
true .
```

- Interpretation: Es gibt ein Tier, das grösser ist als ein Pferd. (Aber wir sind nicht daran interessiert, welches Tier das ist und darum sind wir auch nicht an dessen Ausgabe interessiert)

Zusammengesetzte Terme (compound terms)

- **Zusammengesetzte Terme** haben einen Funktor (functor) (ein Atom) und Argumente (Terme), z.B.:

```
is_bigger(horse, X)
```

**Funktor**
(Atom)

Argumente
(Terme)

- **Weitere Beispiele von zusammengesetzten Termen**

```
bigger(me, you)
```

```
'test print'(hallo)
```

```
f(g(Charlie, _), 77) % compound term inside compound term
```


Atomare Terme, Prädikate und Grundterme

- **Atomare Terme** (atomic terms) = Atome und Zahlen
 - z.B.: 'hallo welt', is_bigger, 666, -7.8
- **Grundterme** (ground terms) = Terme ohne Variablen
 - Praktische Bedeutung: **Fakten** (siehe später)
 - z.B. bigger(me, you), write('bonjour monde')
- **Prädikate** (predicates) = Atome und zus.-ges. Terme
 - Praktische Bedeutung:
 - falls Atome: **Fakten**, z.B. bigger
 - sonst: **Regeln**, z.B.: is_bigger(X, Y) :- bigger(X, Y)

Stelligkeit (Arity) von Prädikaten

- **Stelligkeit** = Anzahl Argumente von einem Prädikat

```
is_bigger(X, Y)
```

 Funktor  2 Argumente, d.h. Stelligkeit 2

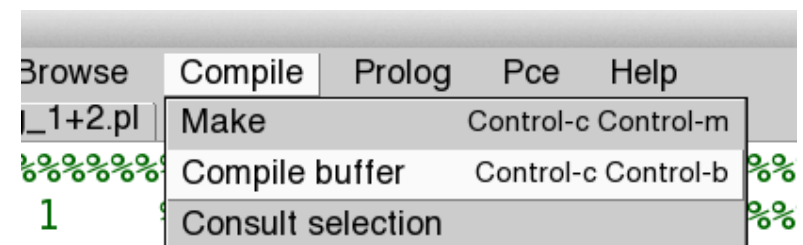
- Prolog behandelt zwei Prädikate mit gleichem Funktor aber mit unterschiedlicher Stelligkeit als zwei verschiedene Prädikate
- In der Prolog-Doku wird die Stelligkeit meist mit Suffix „/“ und danach der entsprechenden Zahl angegeben
 - z.B. `consult/1` oder `working_directory/2`
 - Das verwenden wir auch so! Also z.B.: `is_bigger/2`

Eingebautes Prädikat: consult/1

- (SWI-)Prolog kommt mit ganz vielen eingebauten Prädikaten – diese können wir (natürlich) verwenden!
 - Online Doku: <http://www.swi-prolog.org/pldoc/man?section=builtin>
- Eine Programm-Datei kompilieren: consult/1

```
?- consult(bigger) .
% bigger.pl compiled 0.00 sec, 12 clauses
true.
?-
```

- Liest die Datei bigger oder bigger.pl ein
 - ggf. davor mit working_directory/2 das aktuelle Verzeichnis setzen
 - Kurzform: ['bigger'].
- In SWI-Prolog einfacher mit Menu „Compile – Compile buffer“:



Eingebaute Prädikate: read/1 und write/1

Availability: *built-in*

write(+Term)

[ISO]

Write *Term* to the current output, using brackets and operators where appropriate.

http://www.swi-prolog.org/pldoc/doc_for?object=write/1

- Schreibt das Argument in die Konsole. z.B.:

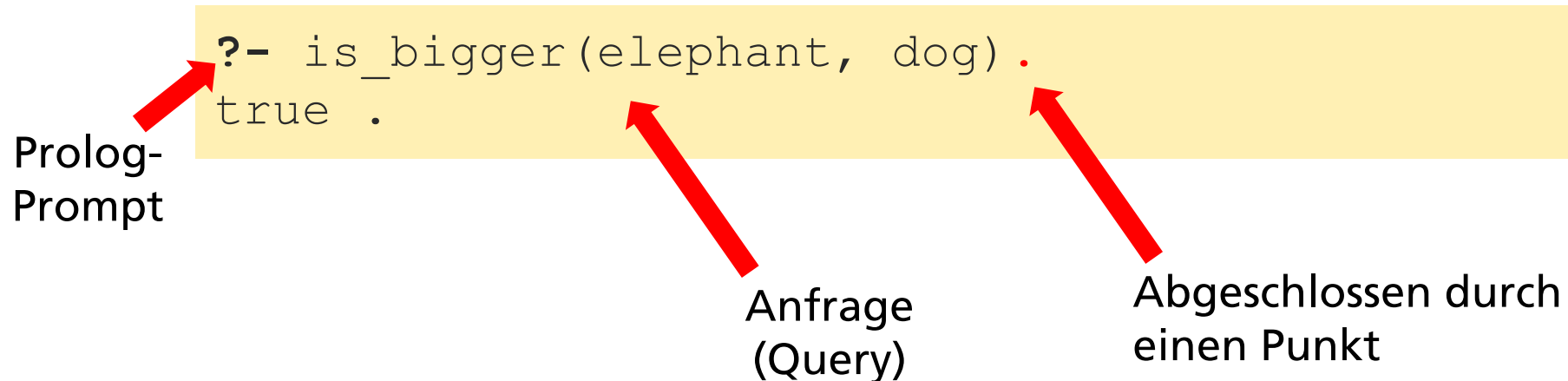
```
?- write('hallo hslu').
hallo hslu
true.
```

- Analog: read/1

```
?- read(X), write(X).
|: 'aha, so geht das!'.           % might look slightly
aha, so geht das!                % different on windows
X = 'aha, so geht das!'.         % (this is from a mac)
```

Prolog Anfragen

- **Anfragen (Queries)** = Prädikate (oder Sequenzen von Prädikaten) gefolgt von einem Punkt
 - Werden beim Prolog-Prompt in der Konsole eingegeben und veranlassen das System zu antworten, z.B.:



Klauseln, Prozeduren & Programme

- **Klauseln (clauses)** = Fakten und Regeln (sind zusammengesetzte Terme)
- **Prozedur (procedure)** = Alle Klauseln zum gleichen Prädikat (d.h. alle Relationen mit gleichem Name [d.h. gleichem Funktor] & gleicher Stelligkeit)
- **Prolog-Programm** = eine Liste von Klauseln

Prolog: Fakten und ...

- Prolog-Programme bestehen aus Fakten und Regeln, die in der aktuellen Wissensdatenbank abgelegt sind
- **Fakten (facts)** = Prädikate gefolgt von einem Punkt
 - Fakten definieren etwas als bedingungslos wahr
 - z.B.

```
bigger(elephant, horse) .  
parent(peter, mary) .
```
 - Fakten sind typischerweise Grundterme (also Terme ohne Variablen)

... Regeln

- **Regeln (rules)** bestehen aus einem Kopf (head) und einem Hauptteil (body), durch :- getrennt
 - Der Kopf einer Regel ist wahr, falls alle Ziele (Prädikate) im Hauptteil als wahr bewiesen werden können, z.B.:

```
grandfather(X, Y) :-           % head
    father(X, Z) ,             % body, goal 1
    parent(Z, Y) .             % body, goal 2
```

- Ziele im Hauptteil werden durch , (Komma) abgetrennt und ganz am Schluss durch . (Punkt) abgeschlossen
 - Bedeutung vom , (Komma): logisches UND, d.h. alle Ziele einer Regel werden ver-UND-et (Fachbegriff: Konjunktion)

Mini-Exkurs: Regeln sind Hornklauseln

- Hornklauseln sind eine spezielle Form von logischen Formeln, welche wie folgt angegeben werden können:

$$(p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow q$$

- \wedge steht für das logische Und (Konjunktion)
 - \rightarrow steht für die logische Implikation
 - Interpretation: Wenn alle Prämissen p_1 bis p_n erfüllt sind, dann lässt sich daraus q ableiten
- Prolog nutzt diese Ableitung: SLD-Resolution (Selective Linear Definite Clause Resolution)
 - Formel von oben in Prolog-Syntax:

$$q \text{ :- } p_1, p_2, \dots, p_n.$$

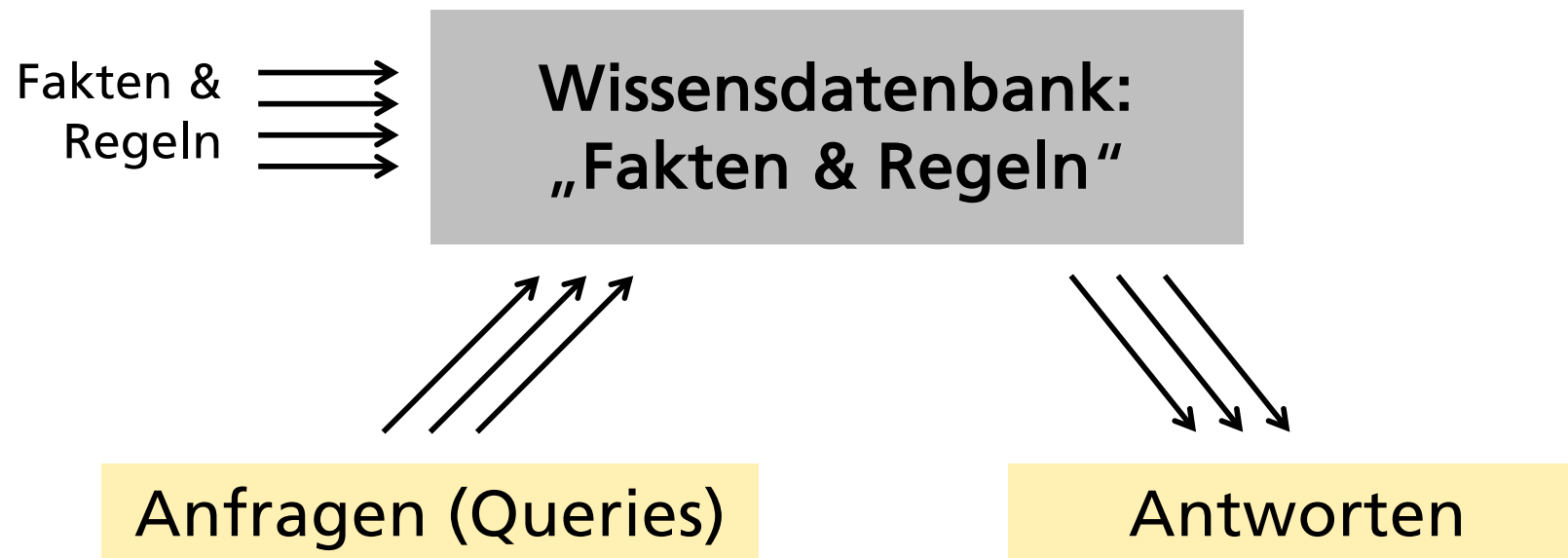


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Matching

Wiederholung: Funktionsweise von Prolog

- Wissensdatenbank (Knowledge Base)
 - Bestehend aus Fakten & Regeln
 - Kann abgefragt werden durch Anfragen (Queries)



...also wie funktioniert nun Prolog?

- Q: Wie werden nun also aufgrund der Fakten und Regeln in der Wissensdatenbank Anfragen an das Prolog-System beantwortet?
- A: Beweissuche mittels Backtracking und Matching

→ Ein wichtiges Konzept ist also **Matching!**
– Siehe nächste Folien...

Matching

- **Allg. Definition:** Zwei Terme matchen, wenn sie identisch sind oder wenn sie durch Ersetzen von Variablen durch andere Terme identisch gemacht werden können
- In Prolog kann mittels dem Gleichheits-Prädikat `=/2` (als Infix-Operator) abgefragt werden, ob zwei Terme matchen. Beispiele:

```
?- eat(tom, burger) = eat(X, burger) .  
X = tom.  
  
?- eat(tom, burger) = eat(X, potato) .  
false.
```

Matching Regel 1: Zwei atomare Terme

- Zwei atomare Terme matchen genau dann wenn sie die gleiche Zahl oder das gleiche Atom sind. z.B.:

```
?- =(tom, tom).      % Predefined predicate =/2
true.

?- tom = tom.        % other style: =/2 as infix operator
true.

?- 'Tom' = tom.      % Watch out: case sensitive!
false.

?- 'tom' = tom.      % =/2 ignores single quotes
true.
```

Matching Regel 2: Variable + Term

- Falls einer der Terme eine Variable ist, dann matchen die beiden Terme und die Variable wird mit dem Wert des zweiten Terms instanziiert. z.B.:

```
?- pia = X.
```

```
X = pia.
```

```
?- sing(pia) = X.
```

```
X = sing(pia).
```

```
?- X = Y.
```

```
X = Y.
```

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

```
X = Y, Y = pia.
```

```
?- X = Y, X = pia, Y = tom.
```

```
false.
```

Matching Regel 3: Zwei zusammengesetzte Terme

- Zwei zusammengesetzte Terme matchen g.d.w.:
 - gleicher Funktor, gleich Stelligkeit
 - alle korrespondierenden Argumente matchen

```
?- meet(drink(pia), eat(X)) = meet(Y, eat(tom)).
```

```
X = tom,
```

```
Y = drink(pia).
```

```
?- meet(X, X) = meet(drink(pia), tom).
```

```
false.
```

```
?- meet(pia, tom) = meet(Y).
```

```
false.
```


Matching (zusammengefasst)

1. Zwei atomare Terme matchen genau dann wenn sie die gleiche Zahl oder das gleiche Atom sind.
2. Falls einer der Terme eine Variable ist, dann matchen die beiden Terme und die Variable wird mit dem Wert des zweiten Terms instanziiert.
3. Zwei zusammengesetzte Terme matchen g.d.w.:
 - gleicher Funktor, gleiche Stelligkeit
 - alle korrespondierenden Argumente matchen

Programmierung mit Matching

■ Zwei Regeln in der Wissensdatenbank

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

■ Beispielabfragen

```
?- vertical(line(point(1,1), point(1,5))).  
true.  
  
?- horizontal(line(point(1,2),point(3,X))).  
X = 2.  
  
?- horizontal(line(point(1,2),P)).  
P = point(_G1155, 2).           % _G1155 means "any number"
```

Kontrollfragen B

Wie wird Prolog auf die folgenden Anfragen antworten?

1. `?- X=tom, X=pia.`
2. `?- 'tom' (pia) = tom('pia').`
3. `?- s(t(g), X) = s(Y, u).`
4. `?- X = f(X).`

Unendliche Terme?

- Was passiert bei folgender Eingabe?

$? - f(A) = A.$

- Gemäss Matching-Regeln müsste Resultat so aussehen: $A = f(f(f(f(f(f(f(f(f(f(...$
 - Also ein unendlicher Term!..
- Antwort von SWI-Prolog: $A = f(A).$

...mh?

Miniexkurs: Der „Occurs Check“

- In der Logik ist Unifikation ein bekanntes Verfahren zur Vereinheitlichung von Ausdrücken
 - Ähnlich wie Matching bei Prolog
- Der grosse Unterschied zwischen Unifikation und Matching: Matching macht keinen „Occurs Check“
 - Occurs Check: Falls eine Variable mit einem Term vereinheitlicht wird, wird zuerst getestet, ob diese Variable in diesem Term vorkommt
 - Wird bei Prolog aus Effizienzgründen meist standardmässig weggelassen, z.B. bei SWI-Prolog
- Unifikation mit Occurs Check kann in SWI-Prolog explizit durchgeführt werden mittels `unify_with_occurs_check/2`

unify_with_occurs_check/2

Availability: *built-in***unify_with_occurs_check(+Term1, +Term2)****[ISO]**

As [=/2](#), but using *sound unification*. That is, a variable only unifies to a term if this term does not contain the variable itself. To illustrate this, consider the two queries below.

```
1 ?- A = f(A).  
A = f(A).  
2 ?- unify_with_occurs_check(A, f(A)).  
false.
```

The first statement creates a *cyclic term*, also called a *rational tree*. The second executes logically sound unification and thus fails. Note that the behaviour of unification through [=/2](#) as well as implicit unification in the head can be changed using the Prolog flag [occurs_check](#).

http://www.swi-prolog.org/pldoc/man?predicate=unify_with_occurs_check/2

- Wir sprechen in Prolog von Matching und nicht von Unifikation
 - Manche Bücher/Quellen trennen das nicht so strikt und verwenden Unifikation in Prolog synonym zu Matching



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Beweissuche & Suchbäume

Beweissuche und Suchbäume

- Nachdem wir nun Matching kennen, haben wir die Voraussetzungen um zu schauen, wie Prolog die Wissensdatenbank durchsucht um zu sehen, ob eine Anfrage erfüllt werden kann
- In andern Worten, wir sind bereit für die **Beweissuche und Suchbäume** in Prolog!

Beispiel: Wissensdatenbank & Anfrage

```
f(a).  
f(b).  
g(a).  
g(b).  
h(b).  
k(X) :- f(X), g(X), h(X). % r1
```

← Wissensdatenbank

```
?- k(Y).
```

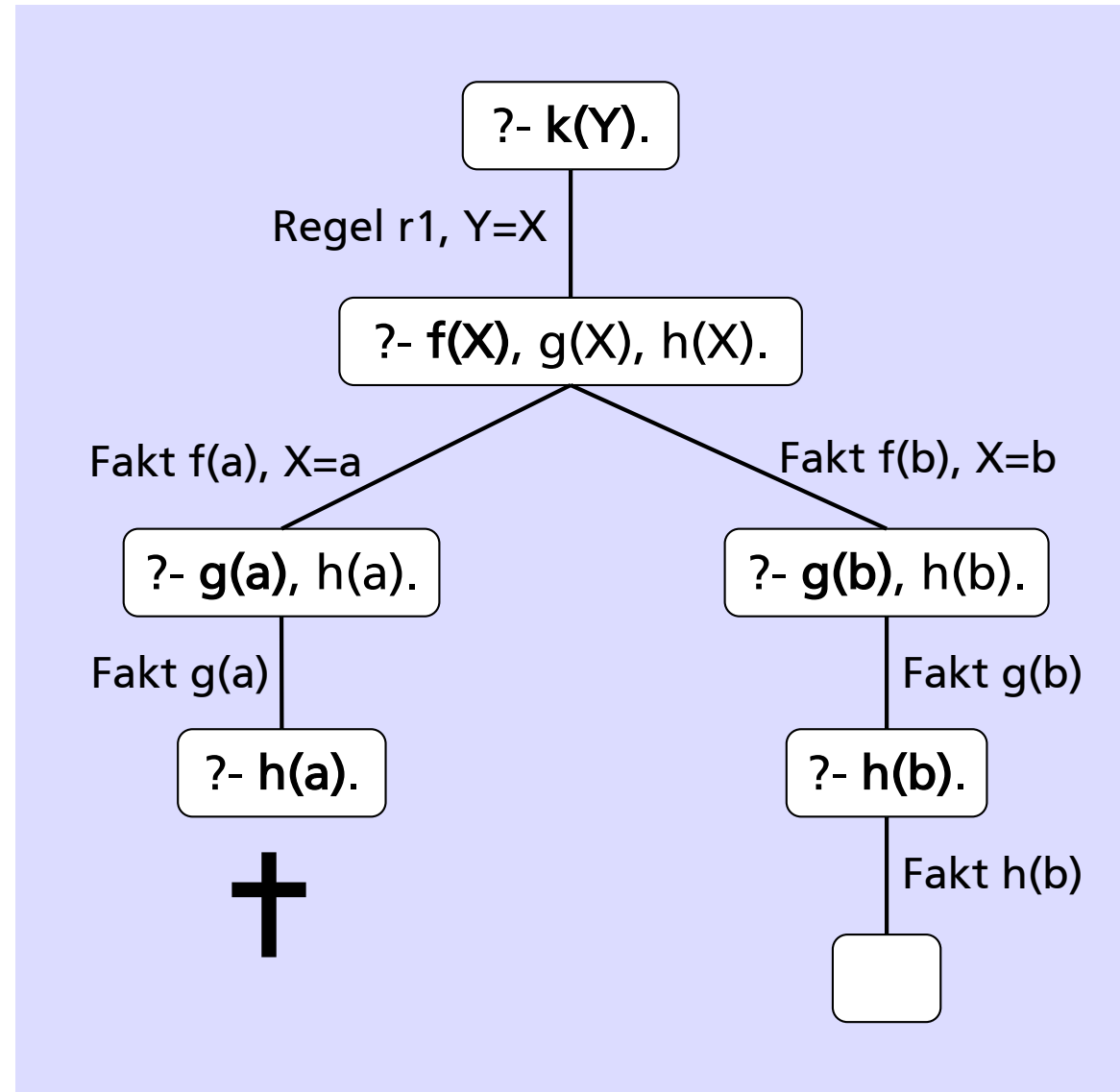
← Anfrage in der
Prolog-Konsole

Beispiel: Beweissuche & Suchbaum

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X). % r1
```

```
?- k(Y).
Y=b.

?-
```



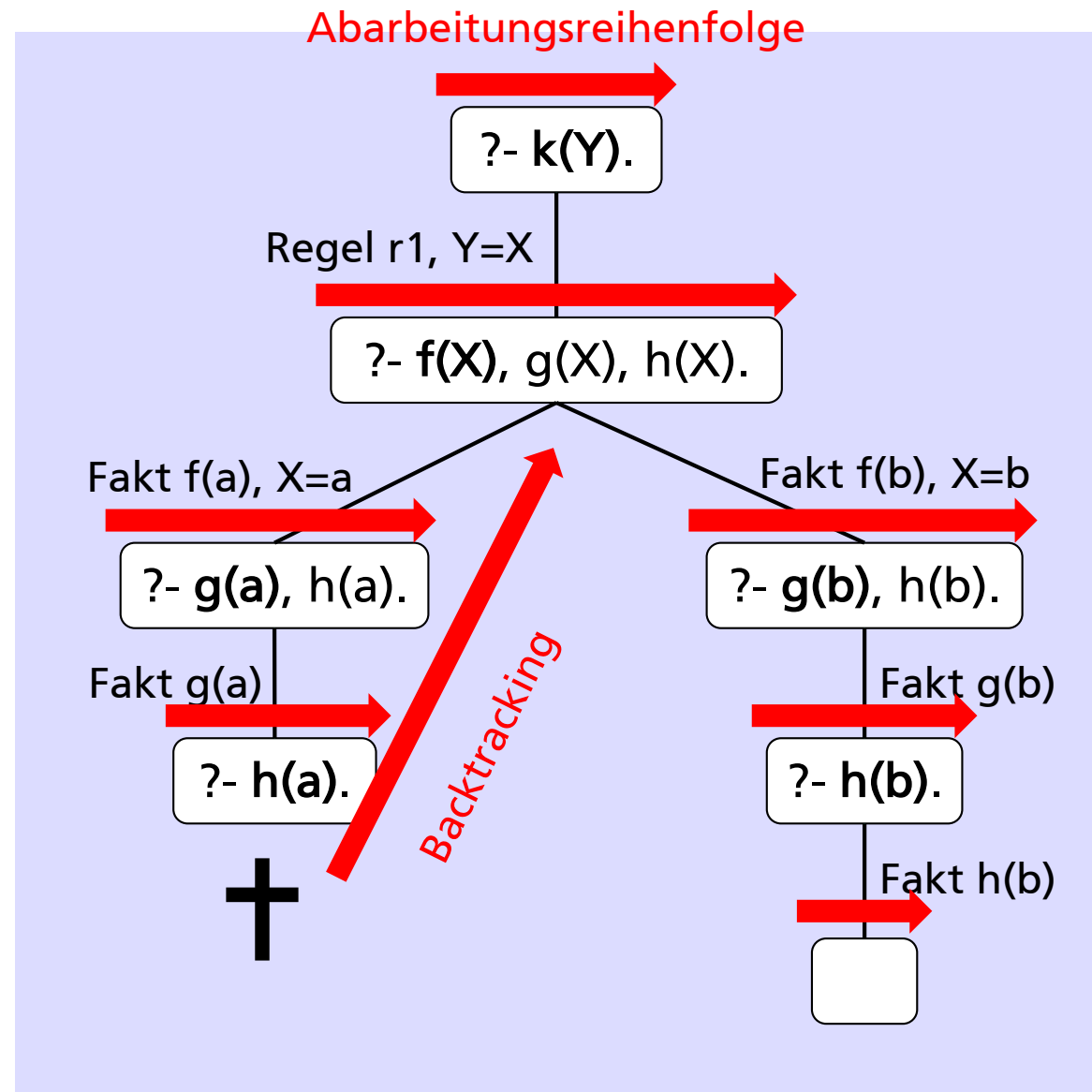
Übersicht: Prolog-Mechanismus

Suchreihenfolge

f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X). % r1

?- k(Y).
Y=b.

?-



Bemerkungen zum Zeichnen von Suchbäumen

- Falls Sie selber Suchbäume zeichnen (z.B. für Übungen oder Modulendprüfungen): Korrekt zeichnen, d.h. insbesondere:
 - Verwendete Regel bzw. Fakten pro Schritt angeben
 - Pro Schritt genau einen Ziel-Term eliminieren (d.h. keine Schritte überspringen oder zusammenfassen!)
 - Pro Schritt Variablen-Matchings angeben
- Siehe Beispiel auf vorangehender Folie
- Live Wandtafel-Demo davon gewünscht?..

Prolog: Beweissuche & Suchbaum

- Bei einer Anfrage an die Wissensdatenbank schaut das System automatisch, ob diese Anfrage aus den aktuellen Fakten & Regeln herleitbar (d.h. beweisbar) ist
- Dabei wird für jeden Term in der Anfrage sequentiell durch die Wissensdatenbank geschaut, mit welchen Klauseln dieser gematched werden kann
- Dadurch entsteht der im vorangehenden Beispiel skizzierte Suchbaum, in welchem mittels Backtracking Lösungen (d.h. ein Beweis) für die Anfrage gesucht werden
- Falls alle Anfrage-Terme (rekursiv) aufgelöst werden können, liefert das System true (ggf. die dazu notwendigen Matchings) zurück, ansonsten false.

Bemerkungen zur Beweissuche

- Die einzelnen Terme der Anfrage werden der Reihe nach „abgearbeitet“, d.h. es wird geschaut, ob sie mit Termen aus der Wissensdatenbank gematched werden können
- Die Anordnung der Fakten und Regeln in der Wissensdatenbank beeinflusst also, wie schnell eine oder mehrere Lösungen gefunden wird, siehe dazu später
- Erinnerung PRG1, ALG
 - Pseudo-Code Backtracking
 - Backtracking ist Tiefensuche

Allgemeiner Backtracking Algorithmus

// Java-Pseudocode:

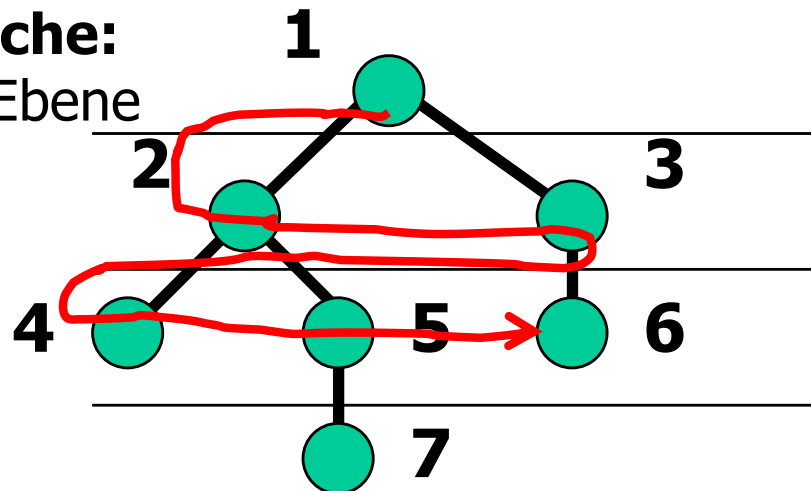
```
boolean findeLoesung(int index, Lsg loesung, ...)  
{  
  while (Es gibt noch neue Teil-Lösungsschritte) {  
    Wähle einen neuen Teil-Lösungsschritt schritt;  
    if (schritt ist gültig) {  
      Erweitere loesung um schritt;  
      if (loesung noch nicht vollständig) {  
        if (findeLoesung(index+1, loesung, ...)) {  
          return true;  
        } else {  
          Mache schritt rückgängig;  
        }  
      } else return true;  
    }  
  } return false;  
}
```

Backtracking ist Tiefensuche

Suche im Lösungsraum:

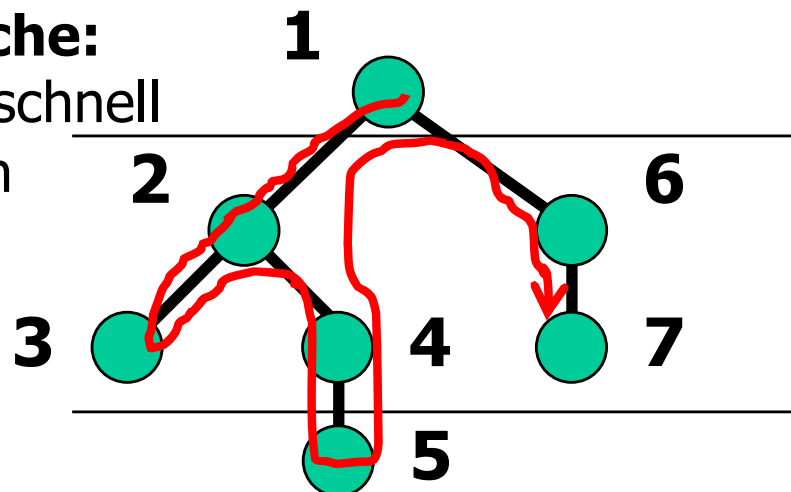
Breitensuche:

Ebene für Ebene

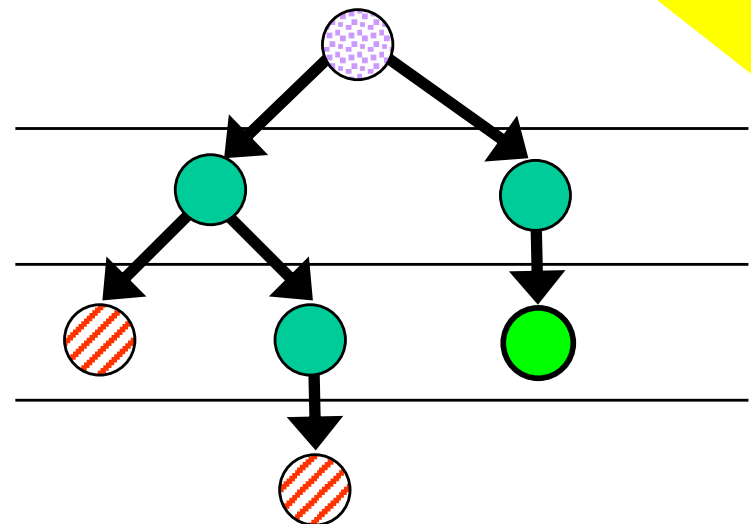


Tiefensuche:

Möglichst schnell
tief suchen



Suchbaum:



→ Lösungsweg

Start

Lösung

Sackgasse

Laufzeitbetrachtungen Tiefsuche

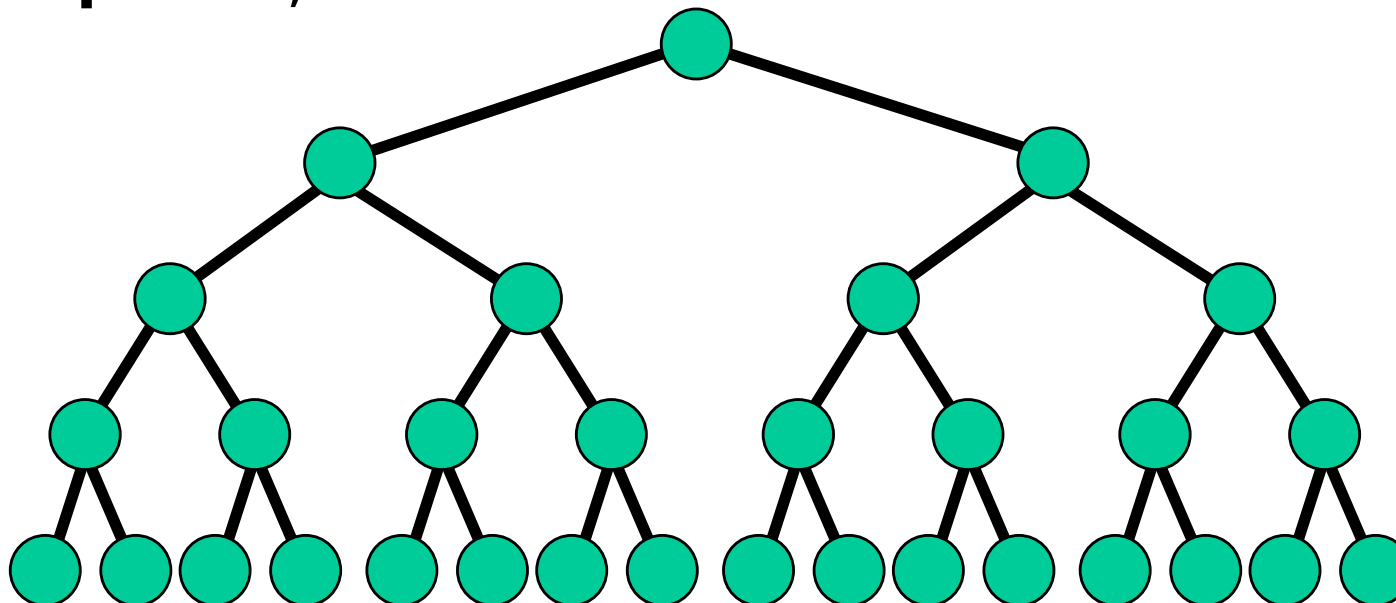
Bei der Tiefsuche werden bei

- max. k möglichen Verzweigungen von jeder Teillösung aus
- in einem Lösungsbaum mit max. Tiefe n im schlechtesten Fall

$$1 + k + k^2 + k^3 + \dots + k^n = \mathbf{O(k^n)}$$

Knoten bzw. Lösungsschritte abgearbeitet.

Bsp.: $k=2, n=4$



Tiefe	Knoten
0	1
1	$k = 2$
2	$k^2 = 4$
3	$k^3 = 8$
n	$k^n = 16$

Beantwortung von Anfragen in Prolog

Zusammenfassend nochmals die drei wesentlichen spezifischen Mechanismen, mit deren Hilfe Prolog Anfragen beantwortet:

- Suche in der Wissensdatenbank von oben nach unten
- Abarbeitung der Abfrage-Klauseln von links nach rechts
- Backtracking zur Erholung von „schlechten“ Pfaden

Prolog: Deklarative & prozedurale Bedeutung

Bisher haben wir in unseren Programmen die Resultate verstanden, ohne genau zu wissen, wie das System genau darauf kam. Es macht daher Sinn, zwischen folgenden zwei Bedeutungsebenen von Prolog-Programmen zu unterscheiden:

- **die deklarative Bedeutung: WAS**
 - Hier geht's ausschliesslich um die im Programm definierten Relationen, diese definieren **was** die Ausgabe vom Programm sein wird
- **die prozedurale Bedeutung: WIE**
 - Hier geht's darum, **wie** diese Ausgabe aus den definierten Relationen abgeleitet werden kann

Zur dekl. Bedeutung von Prolog-Programmen

- Ein grosser Vorteil von Prolog ist die Fähigkeit, viele prozedurale Details selber lösen zu können. Dies ermöglicht es der Programmiererin, deklarative und prozedurale Bedeutung zu trennen
 - Es reicht in Prolog (grundsätzlich), die deklarative Bedeutung anzugeben
 - Das ist von praktischer Bedeutung, da die deklarativen Aspekte von Programmen typischerweise einfacher zu verstehen sind, als die prozeduralen Details
 - D.h. als ProgrammierIn sollte man sich auf die deklarative Bedeutung konzentrieren und sich (wann immer möglich) nicht von prozeduralen Ausführungsdetails ablenken lassen!

Prolog vs. C/Java

- Die beschriebene deklarative Bedeutung macht programmieren in Prolog oft einfacher als in typischen prozedural-orientierten, imperativen Programmiersprachen wie C oder Java
 - „Prolog-Programm = Beschreibung des Problems“
 - Problembeschreibung in Form von Relationen (Fakten und Regeln) -> **deklarativ-logische Programmierung**
 - „Proz.-Programm = Beschreibung der Problemlösung“
 - Schrittweiser, algorithmischer Lösungsweg, basierenden auf einzelnen Befehlen -> **imperative Programmierung**

Prolog: rein deklarativ reicht nicht immer

- Leider ist der deklarative Weg nicht immer ausreichend!
- Speziell in grösseren Prolog-Programmen können aus praktischen Ausführungseffizienzgründen prozedurale Aspekte nicht ganz ignoriert werden
 - Wir werden später darauf eingehen, siehe Endrekursion, Assertions oder Cut-Operator & Negation