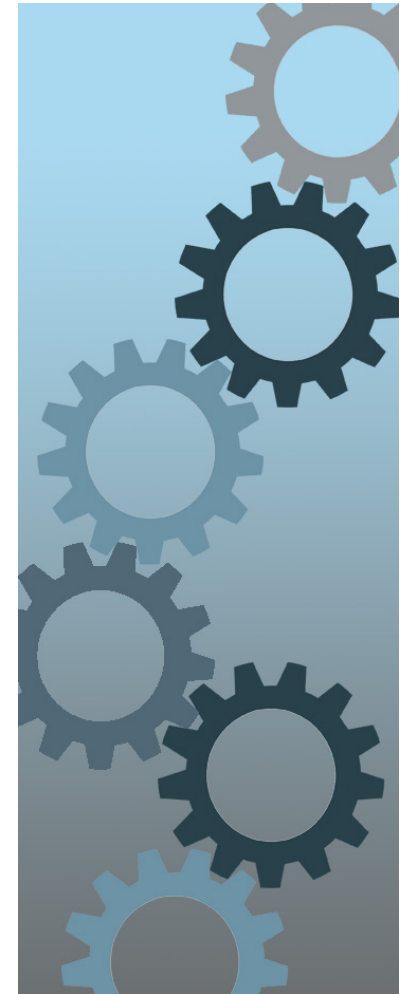


# Programming Concepts & Paradigms

## Prolog 2

Prof. Dr. Ruedi Arnold

[ruedi.arnold@hslu.ch](mailto:ruedi.arnold@hslu.ch)

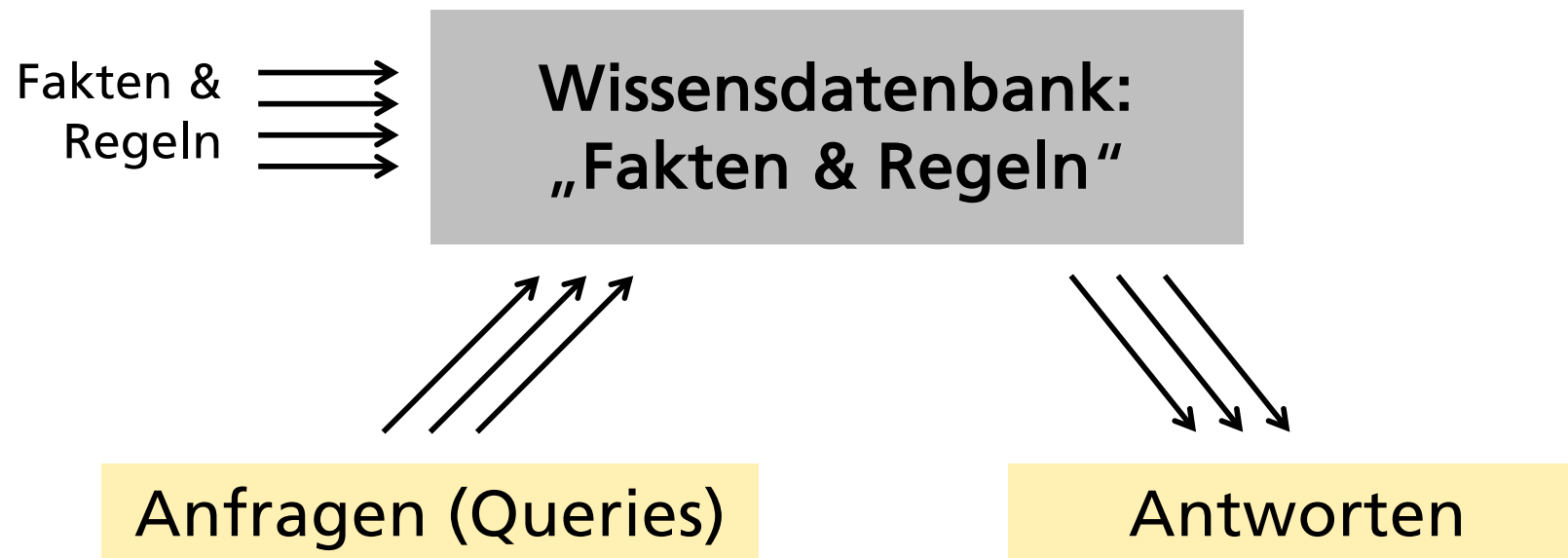


# Übersicht Prolog 2

- Typische Prolog-Probleme
  - Kreuzworträtsel
  - Karten Färben
- Operatoren und Prädikate
  - Arithmetische Operatoren
  - Vergleichsoperatoren
  - Präzedenzen und Typen von Operatoren
- Rekursion
  - Beispiele: Fakultät & Fibonacci

# Wiederholung: Funktionsweise von Prolog

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



# Was für Probleme lassen sich mit Prolog lösen?

- Letztes mal gesehen: Größenrelationen zwischen Tieren mit unseren Prädikaten `bigger/2` und `is_bigger/2`
  - zwei eigene Prädikate, von welchen nur wir die Semantik festlegen und kennen
    - inkl. transitive Hülle
- Ebenfalls gesehen: Prolog arbeitet allgemein mit Relationen zwischen Entitäten
  - Grundidee: Wir beschreiben ein Problem (...deklarativ!) und Prolog löst es dann (prozedural!) für uns

# Typische Prolog-Standardprobleme

- Kreuzworträtsel lösen
- Karten färben
- Zahlenrätsel lösen (siehe später/Übungen)
- Sudoku lösen (siehe später/Übungen)
- Spracherkennung (schauen wir nicht weiter an)
  - Grammatik definieren, korrekte Sätze erkennen, Inhalt „verstehen“ (Semantische Analyse), ...
- Expertensysteme (schauen wir nicht weiter an)
- u.v.a.m.!

...



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

## Beispielproblem: Kreuzworträtsel lösen

# Beispielproblem: Kreuzworträtsel lösen

- Gegeben: 6x4 Felder

- LX = Zeichen in Zelle X  
(ein Buchstabe)

- Erlaubte Wörter (d.h. das „Vokabular“):

- dog, run, top, five, four, lost, mess, unit, baker, forum,  
green, super, prolog, vanish, wonder, yellow

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

- Gesucht: Jede Zelle LX mit einem Buchstaben füllen,  
so dass in allen zusammenhängenden weissen  
Zeilen und Spalten erlaubte Wörter stehen

...wie „sage“ ich das Prolog?

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

- Alle erlaubten Wörter erfassen

- Wir definieren ein neues eigenes Prädikat `word/n`:

```
word(d, o, g) .
```

- Bedeutet für uns: dog ist ein Wort mit drei Buchstaben ( $n = 3$ ), welches im Kreuzworträtsel verwendet werden kann
- Die drei Buchstaben d, o, g und stehen also in dieser Reihenfolge in einer Relation, nämlich: erlaubtes Wort

- Regeln für alle Zeichen LX angeben

- Beschreiben, dass wir 16 Zeichen LX suchen und welche Zeichen LX zusammen ein erlaubtes Wort bilden müssen



# Kreuzworträtsel-Programm in Prolog

```
% words that may be used in the solution
word(d,o,g).      word(r,u,n).      word(t,o,p).      word(f,i,v,e).
word(f,o,u,r).    word(l,o,s,t).    word(m,e,s,s).    word(u,n,i,t).
word(b,a,k,e,r).  word(f,o,r,u,m).  word(g,r,e,e,n).  word(s,u,p,e,r).
word(p,r,o,l,o,g). word(v,a,n,i,s,h). word(w,o,n,d,e,r). word(y,e,l,l,o,w).

solution(L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16) :-
    word(L1,L2,L3,L4,L5),           % Top horizontal word
    word(L9,L10,L11,L12,L13,L14),   % Second horizontal word
    word(L1,L6,L9,L15),             % First vertical word
    word(L3,L7,L11),                % Second vertical word
    word(L5,L8,L13,L16).            % Third vertical word
```

- Damit sind wir fertig: wir haben das Problem beschrieben!
- Wie wird nun dieses Programm aufgerufen?

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

# Kreuzworträtsel lösen: „Prolog in Action“

```
?- solution(L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16) .
L1 = f,
L2 = o,
L3 = r,
L4 = L7, L7 = u,
L5 = m,
L6 = L12, L12 = i,
L8 = L15, L15 = e,
L9 = v,
L10 = a,
L11 = n,
L13 = L16, L16 = s,
L14 = h
false.
```

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	



f	o	r	u	m	
i		u		e	
v	a	n	i	s	h
e				s	

# Kontrollfragen A

1. Wie viele Lösungen für ein Kreuzworträtsel findet unser Programm grundsätzlich?
2. Unser Programm findet für das angegebene Kreuzworträtsel genau eine Lösung. Wie liesse sich das Kreuzworträtsel einfach erweitern, so dass es neu eine zweite Lösung findet? Wie müssten wir dazu das angegebene Programm erweitern? (Machen sie konkrete Vorschläge!)
3. Was würde Prolog antworten, wenn es für ein angegebenes Kreuzworträtsel gar keine Lösung gibt?

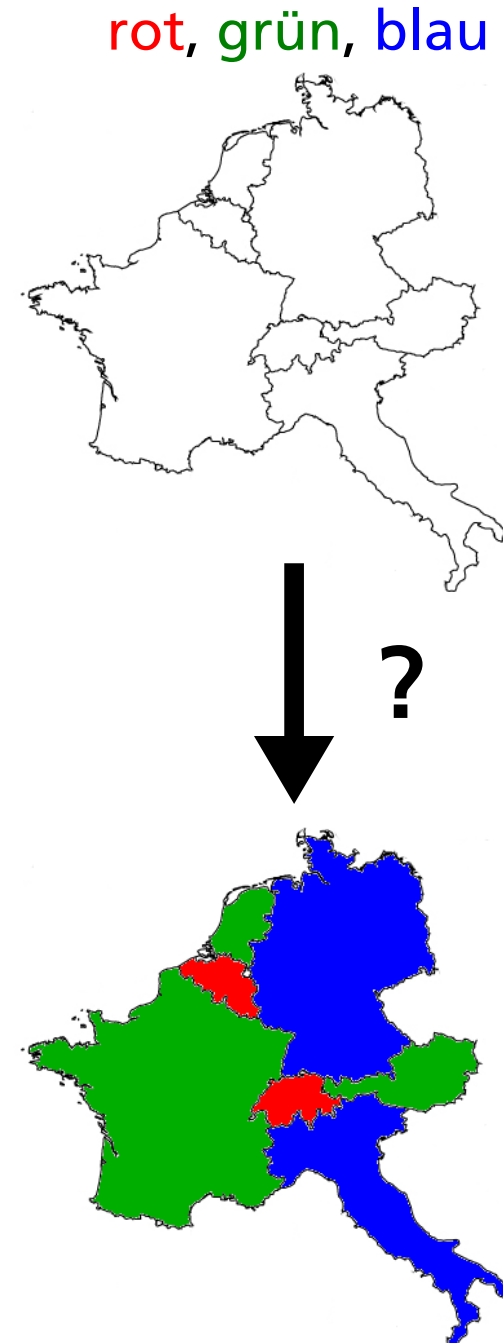


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Beispielproblem: Karten färben

# Beispielproblem: Karten färben

- Gegeben: Karte mit verschiedenen Ländern und 3 verschiedene Farben (rot, grün & blau)
- Gesucht: Färbung, so dass jedes Land eine Farbe hat, und benachbarte Länder jeweils unterschiedliche Farben haben



# Problemmodellierung in Prolog



- Welche Farben können nebeneinander sein?

- Neues Prädikat `n/2` (für „Nachbar“):

```
n(red, green) .
```

- Bedeutet für uns: rot kann neben grün sein

- Welche Länder sind nebeneinander?

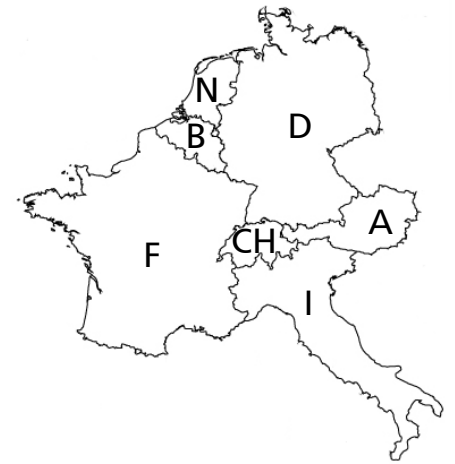
- Modellieren wir mit demselben Prädikat `n/2`

```
n(CH, I) .
```

- Bedeutet für uns: Die Schweiz ist ein Nachbar von Italien

- Technischer: die Variable `CH` ist ein Nachbar der Variable `I`

## Zusätzliche Einschränkung



- All die definierten  $n(X, Y)$  Regeln müssen natürlich gleichzeitig erfüllt sein, daher fügen wir eine neues Prädikat `colors/7` ein, welches die Nachbarschaften von Ländern regelt (durch , getrennt, d.h. „ver-und-et“)
- Um die Anzahl Lösungen klein zu halten, fügen wir als zusätzliche Anforderung ein, dass die Schweiz rot eingefärbt sein muss

`CH = red`

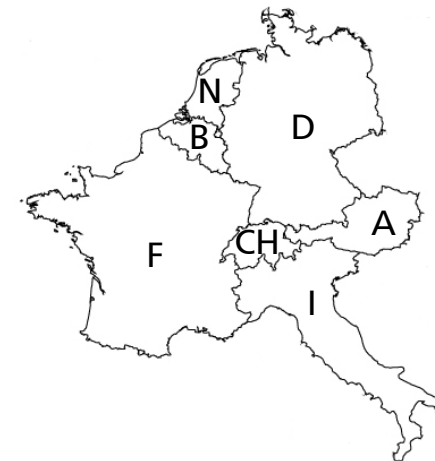
- D.h. der Variablen `CH` muss das Atom `red` zugewiesen werden

# Gesamtes Programm zum Karten färben

```
% Possible pairs of colors of neighboring countries
n(red, green).          n(red, blue).
n(green, red).          n(green, blue).
n(blue, red).           n(blue, green).

% Part of Europe (CH=Switzerland, A=Austria, D=Germany, N=Netherlands...)
colors(CH, A, D, I, F, B, N) :-
    CH = red,                                     % Switzerland must be red
    n(CH, A), n(CH, I), n(CH, F), n(CH, D),      % All neighbors of Switzerland
    n(A, D), n(A, I),                             % All neighbors of Austria (*)
    n(I, F),                                       % All neighbors of Italy (*)
    n(F, B),                                     % All neighbors of France (*)
    n(D, B), n(D, N),                             % All neighbors of Germany (*)
    n(B, N).                                     % All neighbors of Belgium (*)
                                                %(*) = except those already mentioned
```

...und fertig!





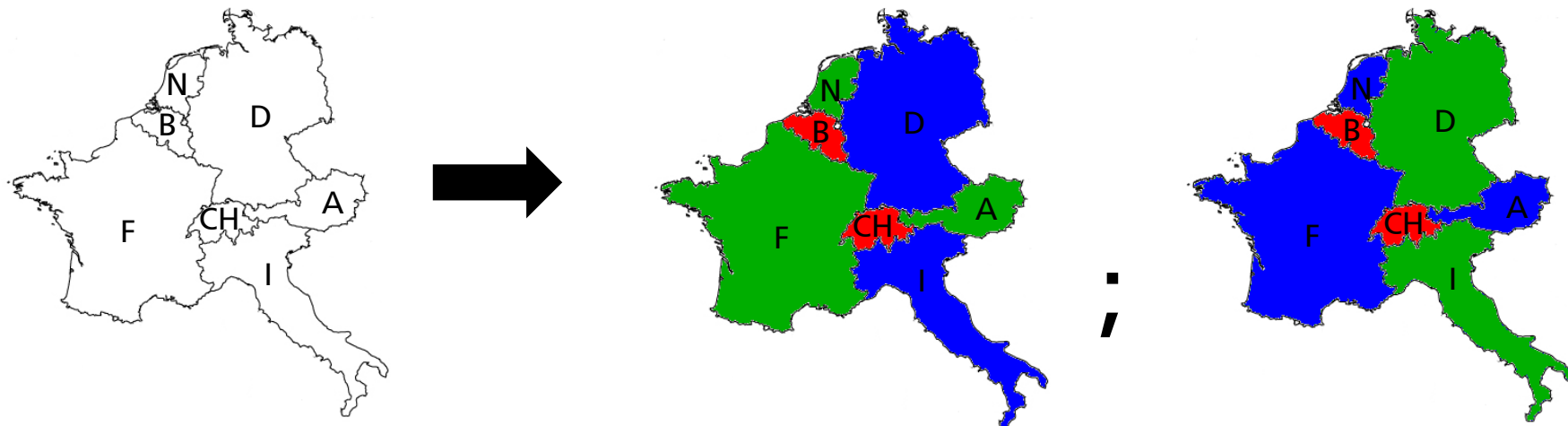
# Prolog: Problembeschreibung reicht

- Damit ist das Problem fertig modelliert 😊
  - Wir haben in diesem Programm das Problem beschrieben (-> deklariert)
    - Deklaration kommt aus dem Lateinischen „declaratio“: Erklärung, Kundmachung, Offenbarung
  - Das Prologsystem „weiss“ damit alles notwendige über unser Färbeproblem, so dass es Lösungen finden kann
    - Lösungssuche wie gesehen mittels Matching und Backtracking
- Vorteil gegeben über imperativer Programmierung, wo Problemlösung schrittweise beschrieben werden muss!

# Problem lösen: Karten färben in Prolog

```
?- colors(CH, A, D, I, F, B, N).  
CH = B, B = red,  
A = F, F = N, N = green,  
D = I, I = blue  
CH = B, B = red,  
A = F, F = N, N = blue,  
D = I, I = green  
false.
```

„;“ gedrückt für „weitere Lösungen“



Geographie-Quizfrage: Welche Länder sind in dieser Karte unterschlagen?  
(Der Einfachheit halber, damit sich das Problem mit 3 Farben lösen lässt! ;-)

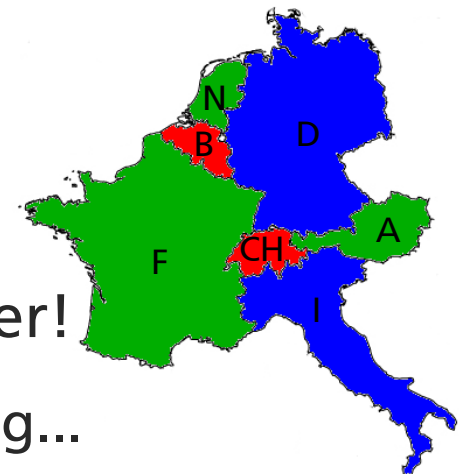
# Hinweise zur Problemmodellierung in Prolog

- Programmierung in Prolog besteht also primär darin, das Problem mithilfe von geeigneten Fakten und Regeln (also Prädikaten) zu modellieren

- Vorteil: Wenn Modell beschrieben, ist das Problem (grundsätzlich) gelöst

L1	L2	L3	L4	L5	
L6		L7		L8	
L9	L10	L11	L12	L13	L14
L15				L16	

- Einschränkung: Effizienz, teilweise ist aus praktischen Gründen die prozedurale Bedeutung von Prolog-Programmen auch wichtig zu verstehen
- Nachteil: Finden einer passenden Modellierung ist nicht unbedingt einfach
  - Erfahrungssache: Übung macht den Meister!
    - Wie z.B. auch bei „guter“ OO-Modellierung...





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

## Operatoren & Prädikate

# Arithmetik und Operatoren in Prolog

```
?- X = 1 + 2.  
X = 1+2.
```

- ...hm, kann Prolog nicht rechnen?
  - So nicht, denn der `=/2`-Operator macht nur ein Matching, Ausdrücke werden nicht automatisch arithmetisch ausgewertet
- Auswertung mittels eingebautem `is/2`-Operator

```
?- X is 1 + 2.  
X = 3.
```

...so hatten wir uns das vorgestellt!

# Der is/2-Operator, bzw. das is/2-Prädikat

**Availability:** *built-in***-Number***is* **+Expr****[ISO]**

True when *Number* is the value to which *Expr* evaluates. Typically, *is/2* should be used with unbound left operand. If equality is to be tested, *==/2* should be used. For example:

```
?- 1 is  
sin(pi/2).
```

Fails! *sin(pi/2)* evaluates to the float 1.0, which does not unify with the integer 1.

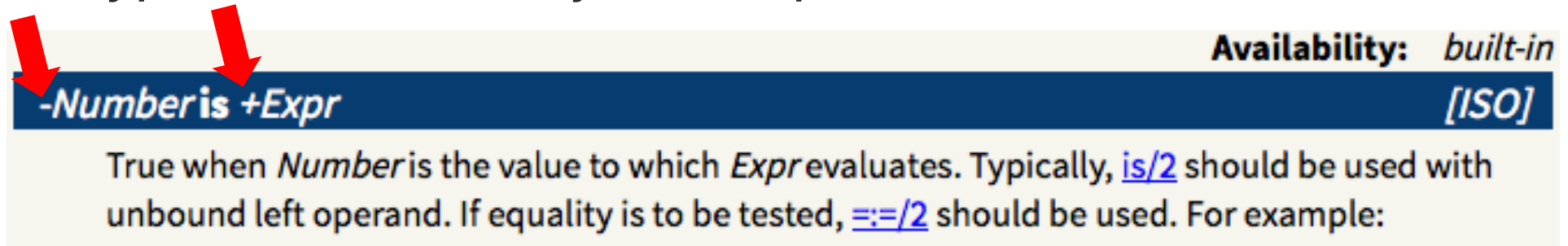
```
?- 1 ==  
sin(pi/2).
```

Succeeds as expected.

- Also: Der eingebaute is/2-Operator erzwingt eine Auswertung, falls die Operanden Zahlen sind

# Gebundene und ungebundene Operanden

- In der Doku von SWI-Prolog stehen bei Prädikaten typischerweise vor jedem Operand -, + oder ?, z.B.:



**Availability:** *built-in*  
*[ISO]*

**-Number is +Expr**

True when *Number* is the value to which *Expr* evaluates. Typically, [is/2](#) should be used with unbound left operand. If equality is to be tested, [=:=/2](#) should be used. For example:

- bedeutet: Operand sollte **ungebunden** (d.h. nicht instanziiert) sein
  - d.h. i.A.: eine Variable („Wert wird durch Prädikat zugewiesen“)
- + bedeutet: Operand sollte **gebunden** (d.h. instanziiert) sein
  - d.h. eine Zahl, ein Atom oder ein gebundener zusammengesetzter Term („Wert wird vom Prädikat ‚gebraucht‘“)
- ? bedeutet: Operand kann gebunden oder ungebunden sein



# Beispiele für - und + bei Operanden

  **-Number is +Expr** Availability: *built-in*  
[ISO]

```
?- X is 7.           % Ok
X = 7.

?- 7 is X.           % Problem
ERROR: is/2: Arguments are not sufficiently instantiated
```

– d.h. is/2 funktioniert nicht „in beide Richtungen“

  **working\_directory(-Old, +New)** Availability: *built-in*

```
?- working_directory(X, '/Users/taarnold/Documents/').
X = '/tmp/'.
```



# Vordefinierte arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Potenz
//	Ganzzahldivision
mod	Modulo (Rest bei Ganzzahldivision)
...	

...und viele weitere! Die ganze Liste hier:

<http://www.swi-prolog.org/pldoc/man?section=functions>

# Beispiele für arithmetische Terme

```
?- D is 5/2.      % division  
D = 2.5.
```

```
?- I is 5//2.     % integer division  
I = 2.
```

```
?- Z is 5 mod 2.  % modulo (remainder of integer div.)  
Z = 1.
```

```
?- P is 2 ** 4.   % power  
P = 16.
```

```
?- C is cos(0).   % cosine  
C = 1.0.
```

```
?- S is sqrt(9).  % square root  
S = 3.0.
```

# Vordefinierte arithmetische Vergleichsoperatoren

>	grösser als
<	kleiner als
>=	grösser-gleich
=<	kleiner-gleich
==	Gleichheit
!=	Ungleichheit

- Hinweis: Diese Operatoren erzwingen die arithmetische Auswertung ihrer beiden Operanden

SWI-Doku: <http://www.swi-prolog.org/pldoc/man?section=arithpreds>

# Beispiele für Terme mit Vergleichen

```
?- 55 > 6.           % is 55 is greater than 6?  
true.  
  
?- 88 =< 77.         % is 88 smaller or equal than 77?  
false.  
  
?- 1 + 2 = 2 + 1.    % Attention: matching!  
false.              % 1+2 does not match 2+1  
  
?- 1 + 2 == 2 + 1.   % is 1+2 equal to 2+1?  
true.  
  
?- 44 \= 42 + 1.     % is 44 unequal to 42+1?  
true.
```

# Prolog: Operatoren vs. Prädikate

- Frage: Was ist der Unterschied zwischen Operatoren und Prädikaten?
  - Beispiele
    - Term mit  $>$ -Operator:  $11 > 7$
    - Term mit  $>/2$ -Prädikat:  $>(11, 7)$
- Antwort: Operatoren und Prädikate sind dasselbe
  - Das sind einfach zwei verschiedenen Schreibweisen. Grundsätzlich gibt es in Prolog nur Prädikate, diese können jedoch auch als Operatoren verwendet werden

## Bsp.: `=/2` und `</2` als Prädikat resp. Operator

- Prädikat `=/2`, schon gesehen in Prolog 1

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

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

- Beispiel mit Prädikat `>/2`

```
?- 3 > 7.           % infix operator style for predicate >/2
false.

?- >(3, 7).         % >/2 predicate, "regular style"
false.
```

# Operator erstellen aus Prädikat: op/3

- Mit dem eingebauten Prädikat op/3 können Prädikate als Operatoren deklariert werden

<b>op(+Precedence, +Type, :Name)</b>	<b>Availability:</b> <i>built-in</i> <i>[ISO]</i>
--------------------------------------	--

- Beispiel (basierend auf is\_bigger/2 aus Prolog 1)

```
?- op(1150, xfx, is_bigger).    % declare new operation
true.

?- elephant is_bigger dog.      % use our new operation
true.
```

- So können wir also z.B. auch eigene Prädikate als Operatoren verwenden! 😊

# Vordefinierte Operatoren (Auswahl)

Präzedenz	Typ	Name
1200	xfx	-->, :-
1200	fx	:-, ?-
1100	xfy	;,
1000	xfy	,
900	fy	\+
700	xfx	<, =, ==, =<, ==, =\=, >, >=, is, ...
500	yfx	+, -, /\, \/ , xor
400	yfx	* , /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	fy	+, -, \

Ganze Tabelle: <http://www.swi-prolog.org/pldoc/man?predicate=op/3>

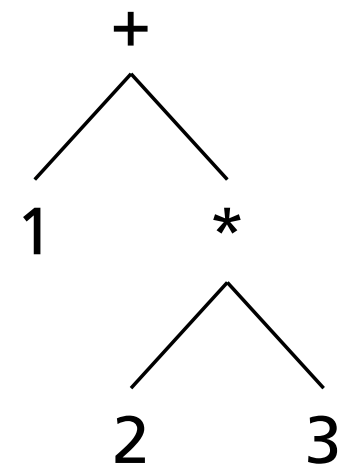


# Präzedenz von Operatoren

- Die Präzedenz (oder Operatorrangfolge) von einem Operator gibt an, wie stark dieser Operator seine Operanden bindet
  - Tiefere Präzedenz = stärkere Bindung
    - d.h. der Operator mit der höchsten Präzedenz ist der Haupt-Funktor von einem gegebenem Term
  - Damit kann Prolog korrekt entscheiden, wie ein Term zu interpretieren ist, d.h. welcher Operator "Vorrang" hat
- In SWI-Prolog wird die Präzedenz als Wert zwischen 0 und 1200 angegeben

## Beispiel zur Präzedenz: \* vor +

- Wie interpretiert Prolog  $1 + 2 * 3$ ?
  - Bedeutet das  $(1 + 2) * 3 = 9$  oder  $1 + (2 * 3) = 7$ ?
- Präzedenz der Operatoren definiert Interpretation!
  - Präzedenz von + ist 500
  - Präzedenz von \* ist 400
  - Das heisst:
    - \* bindet stärker als +
    - + ist der Hauptfunktorktor von diesem Term
      - Siehe Baumdarstellung rechts



# Operator-Typen: Infix, Präfix und Postfix

- Der Typ gibt die relative Reihenfolge von Operator  $f$  und Operanden  $x, y$  an
- Drei Gruppen von Operator Typen:
  - **Infix:**  $xfx, xfy, yfx$ 
    - D.h. Operator  $f$  **zwischen** den beiden Operanden
  - **Präfix:**  $fx, fy$ 
    - D.h. Operator  $f$  **vor** dem Operand
  - **Postfix:**  $xf, yf$ 
    - D.h. Operator  $f$  **nach** dem Operand

## Bsp. Infix-, Präfix- und Postfix-Operatoren

- Infix: is/2-Operator
  - z.B.: X is 7
- Präfix: -/1-Operator
  - z.B.: -77
- Postfix: Fakultätsfunktion (Mathematik)
  - z.B.: 7!
  - **Achtung:** Geht so nicht in Prolog!
    - In Prolog ist das Zeichen ! ganz anders belegt, nämlich als sogenannter Cut mit dem Prädikat !/0, siehe später

## Operanden-Präzedenz: x und y

- Mit x und y wird die **Präzedenz der Operanden** einer Operation f festgelegt:
  - x repräsentiert einen Operanden, dessen Präzedenz strikt kleiner ist als die Präzedenz vom Operator f
  - y repräsentiert einen Operanden, dessen Präzedenz kleiner oder gleich derjenigen vom Operator f ist
- Damit wird festgelegt, wie Ausdrücke mit mehreren gleichen Operatoren ausgewertet werden
  - z.B. bei  $a - b - c$  oder  $1000 / 10 / 10$

# Präzedenz: Links- oder Rechtsassoziativität?

- Wie wird  $1 - 2 - 3$  ausgewertet?
  - Als  $1 - (2 - 3) = 2$  oder  $(1 - 2) - 3 = -4$  ?
  - Ist also „-“ ein rechts- oder linkassoziativer Operator?
    - Mathe: Linksassoziativ, also  $(1 - 2) - 3$   
...ist das bei Prolog auch so?

## Beispiel Operanden-Präzedenz: 1 - 2 – 3

- Wie wird 1 - 2 - 3 ausgewertet?
  - Als  $1 - (2 - 3) = 2$  oder  $(1 - 2) - 3 = -4$  ?
    - ...hm?
- Typ der -/2-Operation: yfx
  - D.h. per Definition darf der linke Operand y eine gleichgrosse Präzedenz haben wie die Operation selbst und der rechte Operand x muss einen kleinere Präzedenz haben als der Operand selbst
    - D.h. der rechte Operand dieser Operation darf selber nicht aus einer -/2-Operation bestehen, denn sonst wäre die Präzedenz vom Operanden gleich gross

# 1 - 2 - 3: Verifikation und Baum-Darstellung

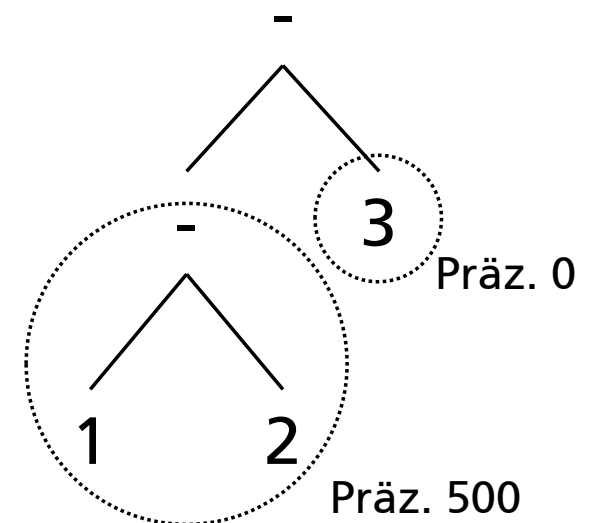
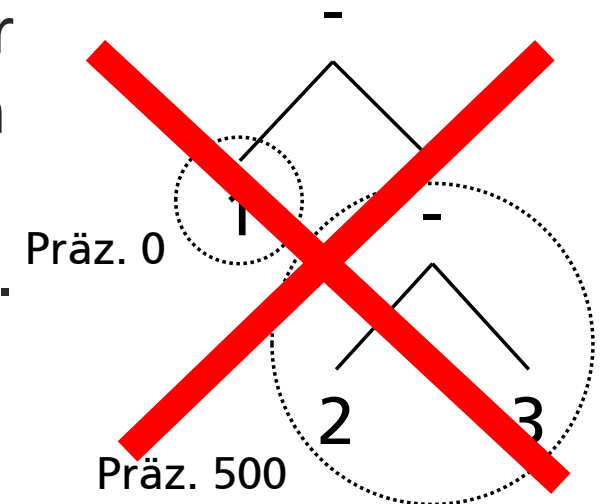
- Nochmals: Der rechte Operand darf hier also selber nicht aus einer -/2-Operation bestehen, denn sonst wäre die Präzedenz vom Operanden gleich gross. D.h. der rechte Operand muss in unserem Beispiel also eine Zahl sein
  - D.h. 1 - 2 - 3 wird in Prolog also ausgewertet als  $(1 - 2) - 3 = -4$

```
?- X is 1 - 2 - 3.
```

```
X = -4.
```

```
?- Y is 1 - (2 - 3) .
```

```
Y = 2.
```





## Recap: Operator- vs. Operanden-Präzedenz

- **Operatoren-Präzedenz:** regelt  $*$  vor  $+$ 
  - Gibt Operatoren-Vorrang bei unterschiedlichen Operatoren an
- **Operanden-Präzedenz:** regelt linksassoziativ ( $yfx$ ) oder rechtsassoziativ ( $xfy$ )
  - Bsp.:  $1 - 2 - 3$  gibt  $-4$
  - Gibt Assoziativität, also Auswertungsreihenfolge bei Ausdrücken mit mehrmals demselben Operator an

# Bsp.: Unklarer Operanden-Vorrang

Präzedenz	Typ	Name
...	...	...
700	<b>xfx</b>	<, =, :=, =<, ==, =\=, >, >=, <b>is</b> , ...

- is/2 ist also vom Typ xfx, d.h. x repräsentiert einen Operanden, dessen Präzedenz strikt kleiner ist als die Präzedenz vom Operator f
- Was passiert bei zweimal demselben xfx-Operator hintereinander?

```
8 ?- X is Y is 44.
```

```
ERROR: Syntax error: Operator priority clash
```

```
ERROR: X i
```

```
ERROR: ** here **
```

```
ERROR: s Y is 44 .
```



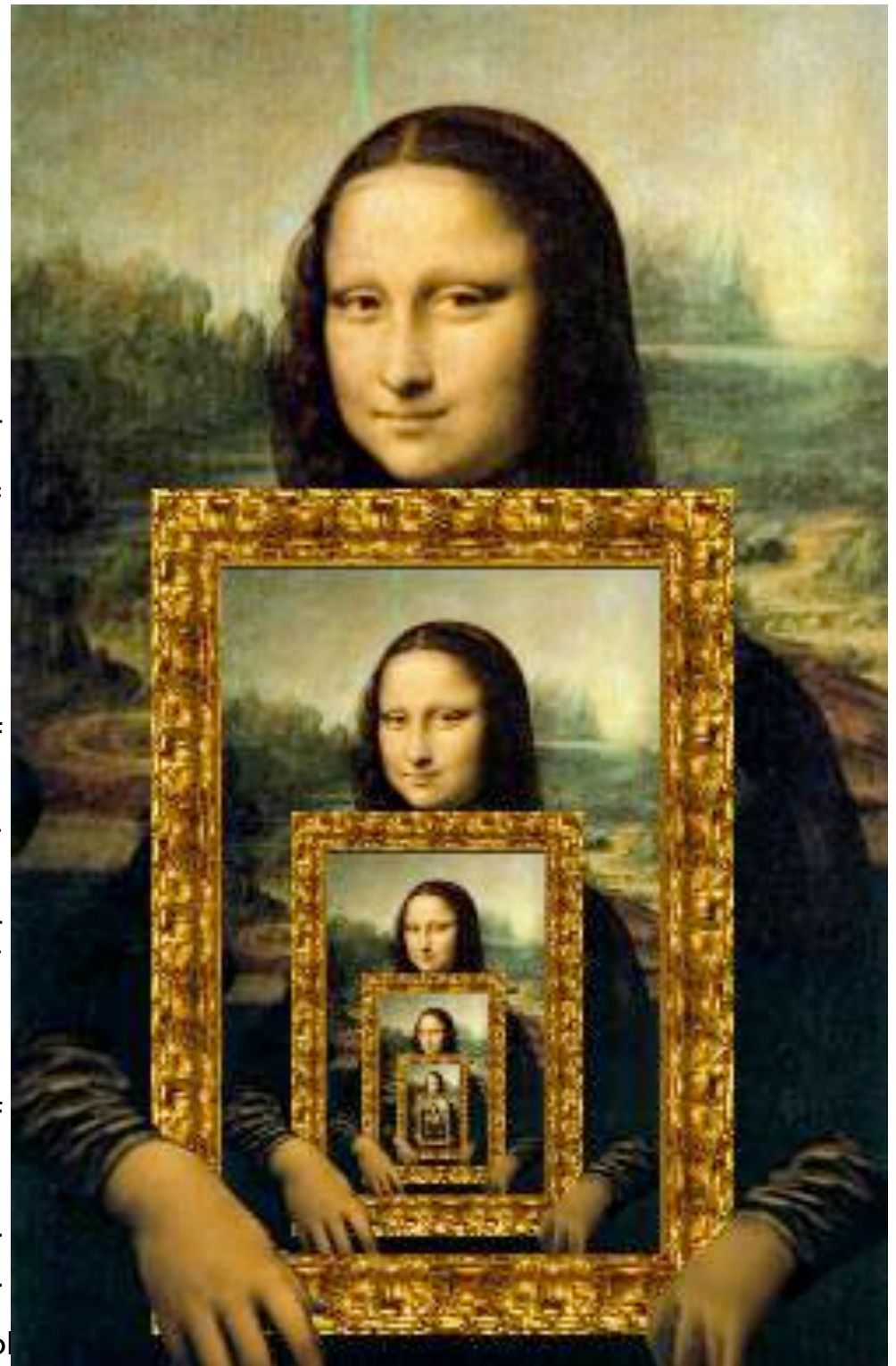
<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Rekursion

# Rekursion

- Prolog Prädikate können rekursiv definiert sein
  - Ein Prädikat ist rekursiv definiert, falls sich eine oder mehrere Regeln in ihrer Definition auf sich selber beziehen
- Rekursion wird sehr viel benutzt in Prolog, werden wir also noch oft sehen!

[http://www.megamonalisa.com/artworks/megamonalisa\\_recursion.jpg](http://www.megamonalisa.com/artworks/megamonalisa_recursion.jpg)





Fraktale  
Hände...

...unendlich  
rekursiv...

und psycho-  
delisch! ;-)



# Einsatz von Rekursion

- Im Prinzip wird Rekursion immer gleich eingesetzt, ein Problem wird in Fälle aufgeteilt, welche zu einer der folgenden beiden Gruppen gehören:
  1. Einfache Fälle oder Grenzfälle, „direkt“ lösbar
  2. Allgemeine Fälle, zu denen die Lösung mithilfe von Lösungen von (einfacheren) Versionen vom gleichen Problem konstruiert werden kann
- Vgl. Modul PRG1 bzw. AD: Diese beiden Fälle entsprechen exakt der Rekursionsbasis und der Rekursionsvorschrift einer Rekursion

## Bsp.: Eigenes rekursives Prädikat is\_bigger/2

- Gesehen in Prolog 1

```
is_bigger(X, Y) :- bigger(X, Y).           % simple case  
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y). % general case
```

- Analyse des rekursiven Prädikates is\_bigger/2
  1. Einfacher Fall: is\_bigger/2 lässt sich direkt auf bigger/2 abbilden, d.h. wir haben entsprechende Fakten
  2. Allg. Fall: is\_bigger/2 wird rekursiv (und transitiv) aufgebaut aus bigger/2 und is\_bigger/2

# Beweissuche & Suchbaum is\_bigger/2

```

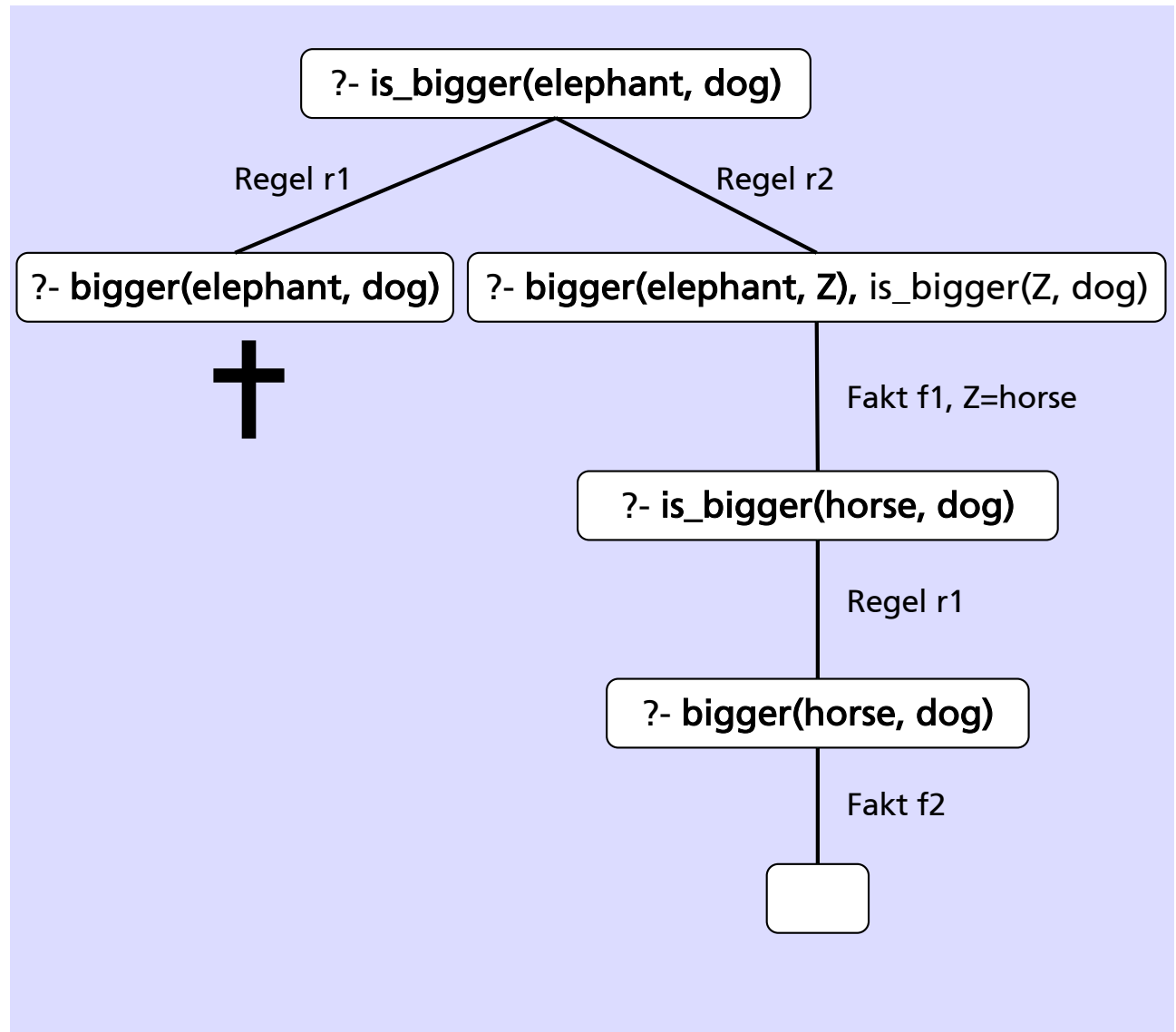
bigger(elephant, horse).      % f1
bigger(horse, dog).           % f2
bigger(horse, sheep).         % f3
is_bigger(X, Y) :- bigger(X, Y). % r1
is_bigger(X, Y) :- bigger(X, Z), % r2
                        is_bigger(Z, Y).
    
```

```

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

```

?-
    
```





# Beispiel: Fakultätsberechnung, Definition von fak/2

- Fakultät  $n!$ , rekursive Definition für  $n \geq 0$ :
  - $0! = 1$
  - $n! = (n-1)! * n$
- Implementierung in Prolog

```
fak(0, 1).                % simple case
fak(N, F) :-              % general case
    N > 0,                % argument test
    N1 is N - 1,          % evaluate N-1
    fak(N1, F1),          % recursive call
    F is N * F1.          % sum up
```

# Berechnungsbeispiele: Fakultät mit fak/2

## ■ Was gibt 5! ?

```
?- fak(5, X) .  
X = 120  
false.
```

## ■ Ist 4! gleich 24?

```
?- fak(4, 24) .  
true.
```

## ■ Andersrum geht leider nicht: $X! = 24 \rightarrow X = ?$

```
?- fak(X, 24) .  
ERROR: >/2: Arguments are not sufficiently instantiated
```

- Grund: >/2-Prädikat braucht gebundene (instanziierte) Operanden
  - D.h.: Regeln mit arithmetischen Prädikaten gelten i.A. nicht „in beide Richtungen“

# Berechnung von 1000!

```
?- fak(1000, X).  
X =  
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610  
197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462  
497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463  
075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085  
379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320  
767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244  
896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421  
88967964624494516076535340819890138544248798495995331910172335556602139450399736280750137837615307127761926849  
034352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476  
088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779911903754031274622289  
988005195444414282012187361745992642956581746628302955570299024324153181617210465832036786906117260158783520751  
516284225540265170483304226143974286933061690897968482590125458327168226458066526769958652682272807075781391858  
178889652208164348344825993266043367660176999612831860788386150279465955131156552036093988180612138558600301435  
694527224206344631797460594682573103790084024432438465657245014402821885252470935190620929023136493273497565513  
958720559654228749774011413346962715422845862377387538230483865688976461927383814900140767310446640259899490222  
221765904339901886018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646  
124960798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281434118  
520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506217095002597389863  
554277196742822248757586765752344220207573630569498825087968928162753848863396909959826280956121450994871701244  
516461260379029309120889086942028510640182154399457156805941872748998094254742173582401063677404595741785160829  
230135358081840096996372524230560855903700624271243416909004153690105933983835777939410970027753472000000000000  
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000  
0000000000000000 .
```

...geht problemlos! (Nicht wie in Java oder C mit Datentyp int!)

d.h.: (SWI-)Prolog rechnet also standardmässig mit beliebig grossen Zahlen ☺

# Beispiel: Berechnung von Fibonacci-Zahlen, fib/2

- Definition Fibonacci Zahlen, für  $n \geq 0$ :

- $F_0 = 0$

- $F_1 = 1$

- $F_n = F_{n-1} + F_{n-2}$

- Fibonacci in Prolog: fib/2

- Bemerkung: Dies ist eine naive Implementierung in Prolog, Optimierungen schauen wir später an (siehe „Endrekursion“ und „Assertions“)

```
fib(0, 0).  
fib(1, 1).  
fib(N, F) :-  
    N > 1,  
    N1 is N - 1,  
    N2 is N - 2,  
    fib(N1, F1),  
    fib(N2, F2),  
    F is F1 + F2.
```

# Berechnungsbeispiele: Fibonacci mit fib/2

- Was ist die 7. Fibonacci-Zahl?

```
?- fib(7, X).  
X = 13 .
```

- Ist die 6. Fibonacci-Zahl 8?

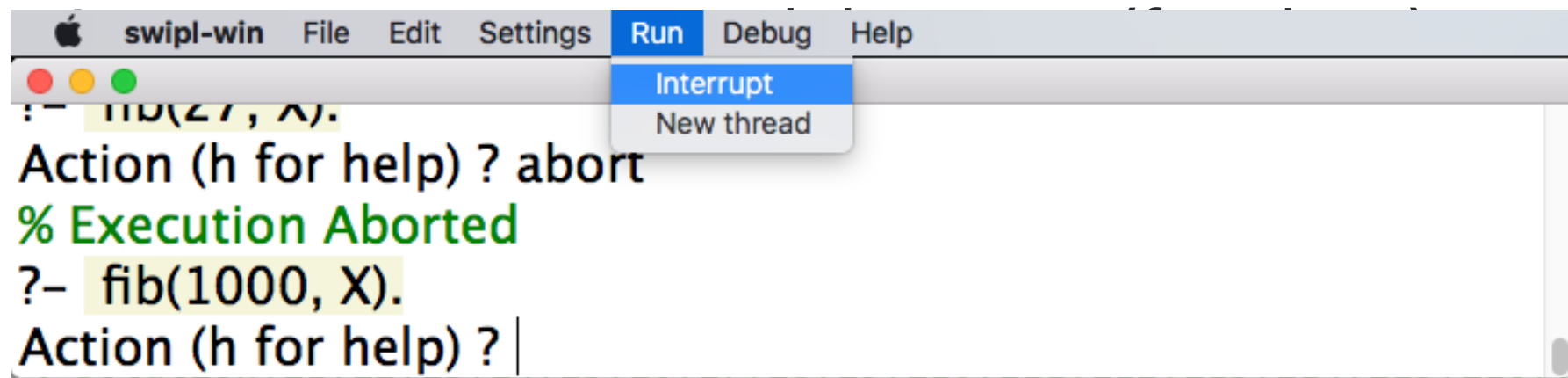
```
?- fib(6, 8).  
true .
```

- Andersrum geht leider auch hier nicht: 8 ist welche Fibonacci-Zahl? (Lösung dazu später mittels CLP)

```
?- fib(X, 8).  
ERROR: >/2: Arguments are not sufficiently instantiated
```

# SWI-Prolog: Anfragen abbrechen?

- 50. Fibonacci-Zahl ist (zu) viel für naives fib/2
  - Rechnet (endlos) lange auf meinem MacBook...
    - MyMac: bis ~22. Fibonacci-Zahl praktisch sofort
  - ...warten auf Stackoverflow? 😊
- Hinweis: Anfragen können abgebrochen werden



## Kontrollfragen B

1. Wie können sie mit dem angegebenen Prädikat `fak/2` herausfinden, was das `X` bei `X! = 720` ist?
2. Wie können sie mittels `fib/2` überprüfen, ob die 8-te Fibonacci-Zahl 21 ist?
3. Was würde passieren, wenn sie beim angegebenen Prädikat `fib/2` die beiden Zeilen „`N1 = ...`“ und „`N2 = ...`“ weglassen und stattdessen die beiden rekursiven Aufrufe direkt machen als `fib(N-1, F1)` und `fib(N-2, F2)`? Funktioniert die Berechnung trotzdem? Begründen sie ihre Antwort.