8. Logische Programmierung

Themen dieses Kapitels:

- Prolog-Notation und kleine Beispiele
- prädikatenlogische Grundlagen
- Interpretationsschema
- Anwendbarkeit von Klauseln, Unifikation
- kleine Anwendungen

Übersicht zur logischen Programmierung

Deklaratives Programmieren:

Problem beschreiben statt Algorithmus implementieren (idealisiert).
Das System findet die Lösung selbst, z. B. Sortieren einer Liste:

```
sort(old, new) <= permute(old, new) ∧ sorted(new)
sorted(list) <= ∀j such that 1 <= j < n: list(j) <= list(j+1)
```

Relationen bzw. Prädikate (statt Funktionen):

```
(a, b)∈ R ⊆ (S x T)
magEssen(hans, salat)
```

Programmkonstrukte entsprechen eingeschränkten **prädikatenlogischen Formeln**

```
∀X,Y,Z: grossMutterVon(X, Z) <= mutterVon(X, Y)∧elternteilVon(Y, Z)
```

Resolution implementiert durch Interpretierer:

Programm ist Menge von PL-Formeln,
Interpretierer sucht Antworten (erfüllende Variablenbelegungen) durch **Backtracking**

```
?-sort([9,4,6,2], X).        Antwort:        X = [2,4,6,9]
```

Datenmodell: strukturierte **Terme mit Variablen** (mathematisch, nicht imperativ);

Bindung von Termen an Variable durch **Unifikation**

Prolog Übersicht

Wichtigste logische Programmiersprache: Prolog (Colmerauer, Roussel, 1971)

Typische Anwendungen: Sprachverarbeitung, Expertensysteme, Datenbank-Management

Ein Programm ist eine **Folge von Klauseln** (Fakten, Regeln, eine Anfrage)
formuliert über Terme.

Fakten

```
mother(mary, jake).
mother(mary, shelley).
father(bill, jake).
```

Regeln

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Anfrage

```
?- parent(X, jake)
```

Antworten:

```
X = mary
X = bill
```

Ein **Interpretierer** prüft, ob Werte an die Variablen so gebunden werden können, dass die Anfrage mit den gegebenen Prädikaten und Regeln erfüllbar ist (Resolution).

Es wird ein **universelles Suchverfahren (Backtracking)** angewendet (Folie GPS-8-7).

Prolog Sprachkonstrukte: Fakten

Fakten geben Elemente von **n-stelligen Relationen** bzw. **Prädikaten** an, z. B.

```
stern(sonne).
stern(sirius).
```

bedeutet, **`sonne`** und **`sirius`** sind Konstante,
sie erfüllen das Prädikat (die 1-stellige Relation) **`stern`**.

Einige Fakten, die Elemente der 2-stelligen Relation **`umkreist`** angeben:

```
umkreist(jupiter, sonne).
umkreist(erde, sonne).
umkreist(mars, sonne).
umkreist(mond, erde).
umkreist(phobos, mars).
```

Fakten können auch mit Variablen formuliert werden:

```
istGleich(X,X).
```

bedeutet in PL: $\forall$ **`X: istGleich(X,X)`**

Prolog hat **keine Deklarationen**. **Namen** für Prädikate, Konstante und Variablen werden **durch ihre Benutzung eingeführt**.
Namen für Konstante beginnen mit kleinem, für Variable mit großem Buchstaben.

Prolog Sprachkonstrukte: Regeln

Regeln definieren **n-stellige Relationen** bzw. **Prädikate**
durch **Implikationen** (intensional), z. B.

```
planet(B) :- umkreist(B, sonne).
satellit(B) :- umkreist(B, P), planet(P).
```

bedeutet in PL:

$\forall$ **B: planet(B)<= umkreist(B, sonne)**
$\forall$ **B,P: satellit(B) <= umkreist(B, P)** $\wedge$ **planet(P)**

In einer Klausel müssen an alle Vorkommen eines Variablennamen dieselben Werte gebunden sein, z. B. **B/mond** und **P/erde**

Allgemein definiert man eine Relation durch **mehrere Fakten und Regeln**.
sie gelten dann alternativ (oder-Verknüpfung)

```
sonnensystem(sonne).
sonnensystem(B) :- planet(B).
sonnensystem(B) :- satellit(B).
```

Man kann Relationen auch **rekursiv definieren**:

```
sonnensystem(sonne).
sonnensystem(X) :- umkreist(X, Y), sonnensystem(Y).
```

Prolog Sprachkonstrukte: Anfragen

Das Prolog-System überprüft, ob eine **Anfrage mit den Fakten und Regeln** des gegebenen Programms (durch prädikatenlogische Resolution) **als wahr nachgewiesen** werden kann.

Beispiele zu den Fakten und Regeln der vorigen Folien:

	Antwort:
`?- umkreist(erde, sonne).`	**yes**
`?- umkreist(mond, sonne).`	**no**

Eine Anfrage bedeutet in PL

```
?- umkreist(mond, B).
∃B: umkreist(mond, B)
```

Wenn die **Anfrage Variablen** enthält, werden **Belegungen** gesucht, mit denen die Anfrage als wahr nachgewiesen werden kann:

	Antworten:
`?- umkreist(mond, B).`	**B=erde**
`?- umkreist(B, sonne).`	**B=jupiter; B=erde; B=mars**
`?- umkreist(B, jupiter).`	**no** (keine Belegung ableitbar)
`?- satellit(mond).`	**yes**
`?- satellit(S).`	**S=mond; S=phobos**

Notation von Prolog-Programmen

Beliebige Folge von **Klauseln**: **Fakten**, **Regeln** und **Anfragen** (am Ende).
Klauseln mit **Prädikaten** `p(`t_1`, ..., `t_n`)`, Terme t_i

Terme sind beliebig zusammengesetzt aus Literalen, Variablen, Listen, Strukturen.

- **Literale** für Zahlen, Zeichen(reihen) `127 "text" 'a'`
- **Symbole** (erste Buchstabe klein) `hans`
- **Variablen** (erste Buchstabe groß) `X Person`
  unbenannte Variable `_`
- **Listen**-Notation: `[a, b, c]  []`
  erstes Element H, Restliste T `[H | T]` wie `H::T` in SML
- **Strukturen**: `kante(a, b)  a - b  datum(T, M, J)`
  Operatoren `kante, -` werden
  ohne Definition verwendet, nicht „ausgerechnet“

Grundterm: Term ohne Variablen, z. B. `datum(11, 7, 1995)`

Prolog ist **nicht typisiert**:

- An eine Variable können beliebige Terme gebunden werden,
- an Parameterpositionen von Prädikaten können beliebige Terme stehen.

Prädikatenlogische Grundlagen

Prädikatenlogische Formeln (siehe Modellierung, Abschn. 4.2):

atomare Formeln p (t_1, ..., t_n) bestehen aus einem Prädikat p und Termen t_i
mit Variablen, z. B. `last([X], X)`
darauf werden logische Junktoren ($\neg \wedge \vee$) und Quantoren ($\forall \exists$) angewandt,

z. B. $\forall X \forall Y$: **`sonnensystem(X)`** $\vee \neg$ **`umkreist(X, Y)`** $\vee \neg$ **`sonnensystem(Y)`**
äquivalent zu
$\forall X \forall Y$: **`sonnensystem(X)`** $\Leftarrow$ **`umkreist(X, Y)`** $\wedge$ **`sonnensystem(Y)`**

Allgemeine PL-Formeln werden auf die 3 Formen von Prolog-Klauseln (Horn-Klauseln) eingeschränkt, z. B.

Prolog-Fakt: **`last ([X], X).`**
PL: $\forall$ **X:** **`last ([X], X).`**

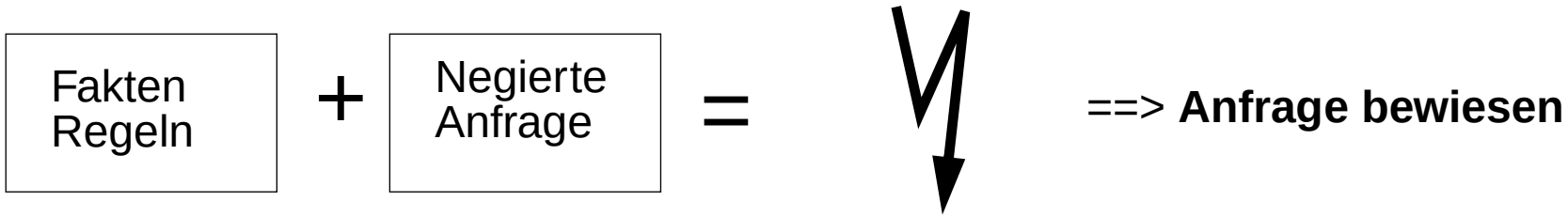
Prolog-Regel: **`sonnensystem(X) :- umkreist(X,Y), sonnensystem(Y).`**
PL: $\forall X \forall Y$: **`sonnensystem(X)<=umkreist(X,Y)`**$\wedge$**`sonnensystem(Y).`**

Prolog-Anfrage: **`umkreist(X, erde), umkreist(X,jupiter).`**
PL: $\exists X$: **`umkreist(X, erde)`** $\wedge$ **`umkreist(X,jupiter).`**

äquivalent zu: $\neg\forall X$ $\neg$**`umkreist(X, erde)`** $\vee \neg$**`umkreist(X,jupiter).`**

Resolution

Resolution führt einen **Widerspruchsbeweis** für eine Anfrage:



==> **Anfrage bewiesen**

Prolog-Anfrage: **`umkreist(X, erde), umkreist(X,jupiter).`**
PL: $\exists$**`X: umkreist(X, erde)`** $\wedge$ **`umkreist(X,jupiter).`**
äquivalent zu: $\neg\forall$**`X`** $\neg$**`umkreist(X, erde)`** $\vee\neg$**`umkreist(X,jupiter).`**

negiert: $\forall$**`X`** $\neg$**`umkreist(X, erde)`** $\vee\neg$**`umkreist(X,jupiter).`**

Die Antwort ist gültig für **alle** zu einem Programm durch induktive Anwendung von Operatoren **konstruierbaren Terme** (Herbrand-Universum, „Hypothese der abgeschlossenen Welt“).

Antwort Ja: Aussage ist mit den vorhandenen Fakten und Regeln beweisbar.
Antwort Nein: Aussage ist mit den gegebenen Fakten und Regeln nicht beweisbar.
Das heißt nicht, dass sie falsch ist.

Daher kann eine Negation, wie in
Formel F gilt, wenn Formel H **nicht** gilt
in Prolog-Systemen nicht ausgedrückt werden.
Der vordefinierte Operator **`not`** ist „nicht-logisch“ und mit Vorsicht zu verwenden.

Interpretationsschema Backtracking

Aus Programm mit Fakten, Regeln und Anfrage spannt der Interpretierer einen **abstrakten Lösungsbaum** auf (Beispiel auf nächster Folie):

Wurzel: Anfrage
Knoten: Folge noch zu verifizierender Teilziele
Kanten: anwendbare Regeln oder Fakten des Programms

Der Interpretierer iteriert folgende Schritte am aktuellen Knoten:

- **Wähle ein noch zu verifizierendes Teilziel** (Standard: von links nach rechts)
  Falls die Folge der Teilziele leer ist, wurde eine Lösung gefunden (success);
  ggf. wird nach weiteren gesucht: backtracking zum vorigen Knoten.
- **Wähle eine auf das Teilziel anwendbare Klausel** (Standard: Reihenfolge im Programm);
  bilde einen neuen Knoten, bei dem das Teilziel durch die rechte Seite der Regel bzw. bei einem Fakt durch nichts ersetzt wird; weiter mit diesem neuen Knoten.
  Ist keine Klausel anwendbar, gibt es in diesem Teilbaum keine Lösung: backtracking zum vorigen Knoten.

Bei rekursiven Regeln, z.b: **`nachbar(A, B) :- nachbar(B, A)`**
ist der **Baum nicht endlich.** Abhängig von der **Suchstrategie terminiert** die Suche dann eventuell **nicht**.

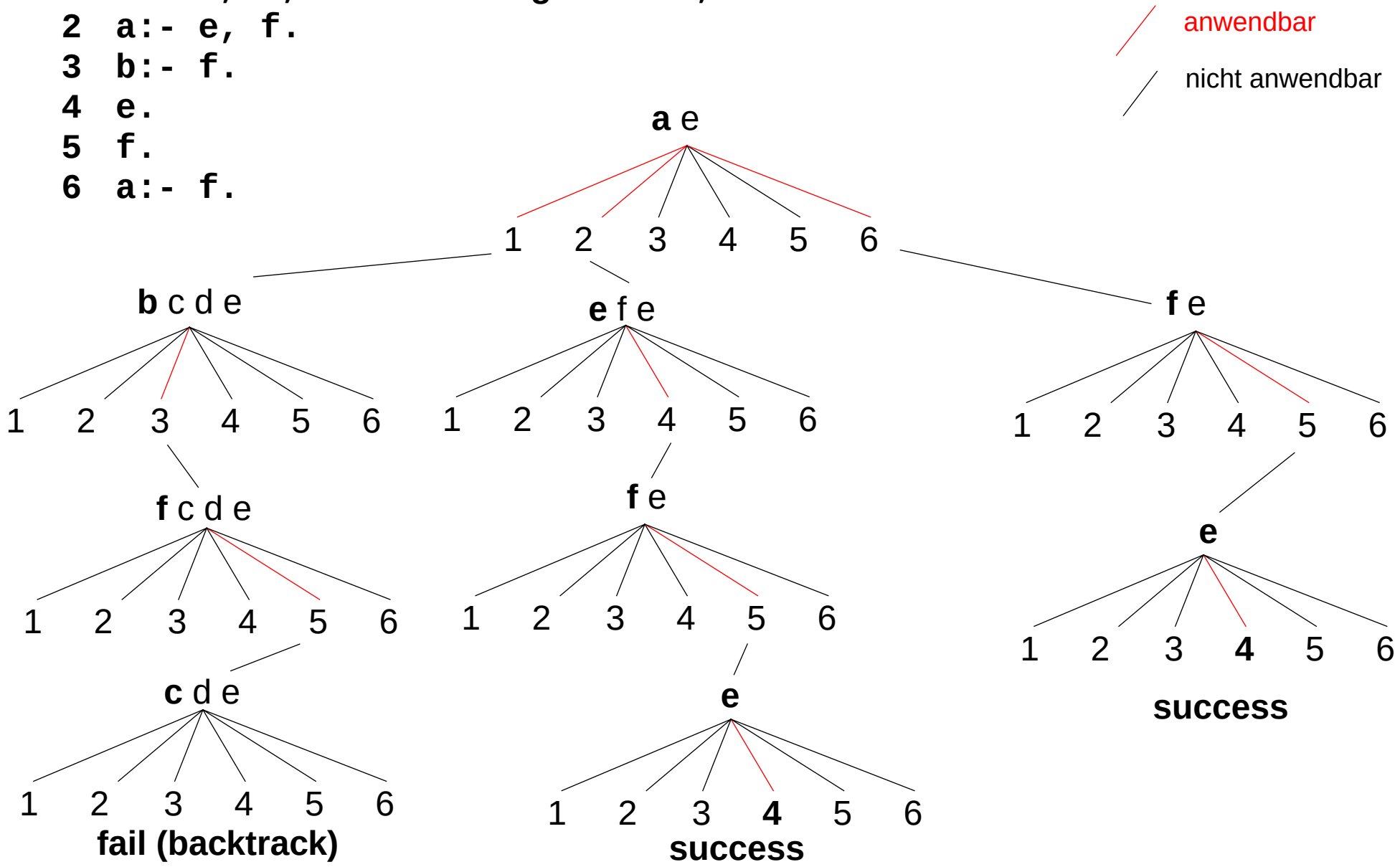
Die Reihenfolge, in der die Wahl (s.o.) getroffen wird, ist entscheidend für die **Terminierung** der Suche und die Reihenfolge, in der Lösungen gefunden werden!

Lösungsbaum Beispiel

Beispiel (**a, b,** ... stehen für Prädikate; Parameterterme sind hier weggelassen):

```
1  a:- b, c, d.
2  a:- e, f.
3  b:- f.
4  e.
5  f.
6  a:- f.
```

Anfrage: ?- a, e



Anwendung von Klauseln

In Klauseln werden **Terme als Muster** verwendet.
Darin vorkommende Variablennamen müssen konsistent an Terme gebunden werden:

`last([X], X).` `[X]` Muster für eine einelementige Liste
`heuer(T, M, datum(T, M, 2013)).` Muster für ein `datum` mit bestimmten Teiltermen

Eine Klausel (Fakt oder linke Seite einer Regel) ist auf ein Teilziel anwendbar,
wenn es einen **Unifikator** gibt, der die **Parameterterme der Klausel und des Teilziels paarweise gleich macht**:

Fakt: `heuer(T, M, datum(T, M, 2013)).`
Anfrage: `?-heuer(12, 7, Z).` Unifikator: `[T/12,M/7, Z/datum(12,7,2013)]`

Fakt: `heuer(T, M, datum(T, M, 2013)).`
Anfrage: `?-heuer(X, Y, datum(14, 7, 2013).`
Unifikator: `[X/14,T/14,Y/7, M/7]`

Regel: `last([_|T], Y):- last(T, Y).`
Teilziel: `last([2,3], Z)` Unifikator: `[T/[3], Y/Z]`

Fakt: `last([X], X).`
Teilziel: `last([2,3], Z)` nicht unifizierbar, also nicht anwendbar

Wird die Klausel angewandt, werden die **Variablen gemäß Unifikator gebunden**.

Unifikation

siehe Modellierung, Kap. 3.1
Term: Formel bestehend aus Literalen, Variablen, Operatoren, Funktoren; z. B. **`X + f(2*Y)`**

Substitution `s = [X1/e1, ..., Xn/en]` angewandt auf `T`, geschrieben **`T s`** bedeutet:
alle Vorkommen der Variablen X_i in **`T`** werden gleichzeitig durch den Term e_i ersetzt.
z. B. **`Y+Y [Y/3*Z]`** ergibt **`3*Z+3*Z`**

Unifikation: Allgemeines Prinzip: Terme durch Substitution gleich machen.
gegeben: zwei Terme T_1, T_2
gesucht: eine Substitution **`U`**, sodass gilt T_1 **`U`** = T_2 **`U`**. Dann ist **`U`** ein **Unifikator** für **`T1`** und **`T2`**.

Beispiele:

```
datum(T, M, 2011)
datum (14, 7, 2011)

U =  [T/14, M/7]
```

```
X+f(2*g(1))
3+f(2*Y)

U = [X/3, Y/g(1)]
```

```
f(h(a, b), g(Y),    V)
f(X,       g(h(a,c)), Z)
```

allgemeinste Unifikatoren:

```
Ua = [X/h(a,b), Y/h(a,c), V/Z]

Ua = [X/h(a,b), Y/h(a,c), Z/V]
```

nicht-allgemeinster Unifikator,
unnötige Bindungen an V und Z:

```
U = [X/h(a,b), Y/h(a,c), V/a, Z/a]
```

Rekursive Anwendung von Klauseln

Variable sind lokal für jede Anwendung einer Klausel.

Bei **rekursiven Anwendungen** entstehen **neue lokale Variable.**

Mehrfache Auftreten einer Variable stehen für denselben Wert.

Beispiel: mit folgenden Klauseln

```
(1)last([X], X).
(2)last([_|T], Y):- last(T, Y).
```

wird die Anfrage berechnet:

```
      ?-last([1,2,3], Z).
(2)     last([_|T1],  Y1):- last([2,3], Z).
(2)                         last([_|T2],Y2):- last([3], Z).
(1)           T1 = [2,3]                      last([X], X). bindet Z=3
              Y1 = Z          T2 = [3]              X = 3
                              Y2 = Z                X = 3 = Z
```

Beispiel: Wege im gerichteten Graph

Das folgende kleine Prolog-Programm beschreibt die Berechnung von Wegen in einem gerichteten Graph.

Die Menge der gerichteten Kanten wird durch eine Folge von Fakten definiert:

```
kante(a,b).
kante(a,c).
...
```

Die Knoten werden dabei implizit durch Namen von Symbolen eingeführt.
Die Relation **weg(X,Y)** gibt an, ob es einen Weg von **X** nach **Y** gibt:

```
weg(X, X).                          Weg der Länge 0
weg(X, Y):-kante(X, Y).             Weg der Länge 1
weg(X, Y):-kante(X, Z), weg(Z, Y).    weitere Weg e
```

Anfragen:

?-weg(a,c).	prüft, ob es einen Weg von **a** nach **c** gibt.
?-weg(a,X).	sucht alle von a erreichbaren Knoten.
?-weg(X,c).	sucht alle Knoten, von denen **c** erreichbar ist .

Beispiel: Symbolische Differentiation

Das folgende Prolog-Programm beschreibt einige einfache Regeln zur Differentiation.
Sie werden auf Terme angewandt, die Ausdrücke beschreiben, und liefern die Ableitung in Form eines solchen Terms, z. B. **`?-diff(2*x,x,D).`** liefert z. B. **`D = 2*1+x*0.`**
Mit weiteren Regeln zur Umformung von Ausdrücken kann das Ergebnis noch vereinfacht werden.

In Prolog werden Ausdrücke wie **`2*x` nicht ausgewertet** (sofern nicht durch **`IS`** explizit gefordert), sondern als Struktur dargestellt, also etwa **`*(2, x)`**.

Prolog-Regeln zur Symbolischen Differentiation:

```
diff(X, X, 1):- !.
diff(T, X, 0):- atom(T).
diff(T, X, 0):- number(T).

diff(U+V, X, DU+DV):- diff(U, X, DU), diff(V, X, DV).

diff(U-V, X, DU-DV):- diff(U, X, DU), diff(V, X, DV).

diff(U*V, X, (U*DV)+(V*DU)):- diff(U, X, DU), diff(V, X, DV).

diff(U/V, X, ((V*DU)-(U*DV))/V*V):- diff(U, X, DU), diff(V, X, DV).
```

Falls die erste Regel anwendbar ist, bewirkt der **Cut** (**!**), dass bei beim Backtracking keine Alternative dazu versucht wird, obwohl die nächsten beiden Klauseln auch anwendbar wären.

Erläuterungen zur Symbolischen Differentiation

1. Hier werden Terme konstruiert, z. B. zu **`2*x`** der Term **`2*1+x*0`**
   Ausrechnen formuliert man in Prolog durch spezielle IS-Klauseln:
   `dupl(X,Y):- Y IS X*2.` **`X`** muss hier eine gebundene Variable sein.

2. Problemnahe Beschreibung der Differentiationsregeln, z. B. Produktregel:

$$\frac{d(u*v)}{d\,x} = u * \frac{d\,v}{d\,x} + v * \frac{d\,u}{d\,x}$$

3. **`diff`** ist definert als Relation über 3 Terme:
   `diff` (abzuleitende Funktion, Name der Veränderlichen, Ableitung)

4. Muster in Klauselkopf legen die Anwendbarkeit fest, z. B. Produktregel:
   `diff(U*V, X, (U*DV)+(V*DU)):-` ...

5. Regeln 1 - 3 definieren:

$$\frac{d\,x}{d\,x} = 1 \qquad \frac{d\,a}{d\,x} = 0 \qquad \frac{d\,1}{d\,x} = 0$$

   !-Operator (Cut) vermeidet falsche Alternativen.

6. **`diff`** ist eine Relation - nicht eine Funktion!!
   ?-**`diff(a+a,a,D).`** liefert **`D = 1 + 1`**
   ?-**`diff(F,a,1+1).`** liefert **`F = a + a`**

Beispielrechnung zur Symbolischen Differentiation

```
?- diff(2*y, y,  D)
   diff(U*V, X1,(2*DV)+(y*DU)):- diff(2, y, DU),   diff(y, y, DV)
                                 diff(T1, X2, 0)   diff(X3, X3, 1)
                                     :-number(2)       :- !
                                       success           success
```

liefert Bindungen **DU=0 DV=1 D=(2*1)+(y*0)**

Das Programm kann systematisch erweitert werden, damit Terme nach algebraischen Rechenregeln vereinfacht werden, z. B.

```
simp(X*1, X).   simp(X+0, X).
simp(X*0, 0).   simp(X-0, X).
...
```

So einsetzen, dass es auf alle Teilterme angewandt wird.

Zusammenfassung zum Kapitel 8

Mit den Vorlesungen und Übungen zu Kapitel 8 sollen Sie nun Folgendes können:

- Kleine typische Beispiele in Prolog-Notation lesen, verstehen und schreiben
- Interpretationsschema und prädikatenlogische Grundlagen verstehen
- Unifikation zum Anwenden von Klauseln einsetzen