

Proseminar Programmiersprachen

# Prolog

Johannes Weißl

TU München

15. November 2006

- 1 Einführung
- 2 Grundlegende Sprachelemente
- 3 Weiterführende Konzepte
- 4 Ausführung
- 5 Programm

- **Prolog** („**P**rogramming in **L**ogic“): entwickelt 1972 von Alain Colmerauer und Robert Kowalski
- Geringe Beachtung, u.a. wegen Verbreitung von *Lisp* und Misserfolgen mit früheren logischen Sprachen wie *Planner*
- Aufschwung durch FGCS (Fifth Generation Computer Systems project) in Japan in den 80ern
- Verbreitung: v.a. Computerlinguistik, Künstliche Intelligenz, Expertensysteme und Computerbeweise

## Imperative Sprachen

- Orientierung an *Von-Neumann-Architektur*
- Nur schrittweise Abstraktion

## Funktionale oder Logische Sprachen

- Deklarativ: nicht *wie*, sondern *was*
- Abstrakte Grundlage:
  - Mathematische Funktionen,  $\lambda$ -Kalkül
  - Prädikatenlogik, Logische Programmierung

- Grundkonzept:

Programm = Menge von Axiomen

Ausführung = Beweis eines Zielausdrucks  
aus dem Programm

- Prolog: Umsetzung des Konzepts

- Beachtung von Effizienz

- Nicht-Logische Elemente für reale Anwendungen

# Grundlagen: Fakten

- Prolog-Programm: Sammlung aus **Fakten** und **Regeln**
- Beispiele für **Fakten**:
  - Marge ist weiblich.
  - Homer ist ein Elternteil von Lisa.
- Formal:

```
female(marge).  
parent(homer,lisa).
```

- Begriffe:
  - Atome**: z.B. marge, homer, lisa
  - Prädikate** (Relationen): z.B. female, parent

## Beispielprogramm $P_S$ : Die Simpsons

```
female(marge).
female(lisa).
female(maggie).
male(grandpa).
male(homer).
male(bart).
parent(grandpa,homer).
parent(marge,bart).
parent(marge,lisa).
parent(marge,maggie).
parent(homer,bart).
parent(homer,lisa).
parent(homer,maggie).
```

- female/1, male/1: einstelliges Prädikat
- parent/2: zweistelliges Prädikat
- Allgemein:  $f(t_1, t_2, \dots, t_n)$   
 $n$ : **Stelligkeit** —  $f$ : **Funktor** —  $t_1, \dots, t_n$ : **Argumente**

Anfragen an das Programm über Prolog-System:

## Beispiel

female(lisa)?

*yes*

female(bart)?

*no*

female(edna)?

*no*

parent(bart,homer)?

*no*

parent(homer,bart)?

*yes*

# Variablen in Anfragen

- Logische **Variable**: Platzhalter für einen Wert  
(**kein** Behälter für Speicherobjekte)
- Zur Unterscheidung: Großbuchstabe am Wortanfang

## Beispiel

parent(marge, **X**)?

*X = bart ;*

*X = lisa ;*

*X = maggie*

parent(**Y**,maggie)?

*Y = homer ;*

*Y = marge*

- **Substitution:** Menge  $\theta = \{x_1 \mapsto t_1, \dots, x_i \mapsto t_i\}$   
Anwenden auf Ausdruck: Ersetzen aller Vorkommnisse

## Beispiel

$$\begin{array}{l} t = \text{parent}(Y, \text{maggie}) \\ \theta = \{Y \mapsto \text{homer}\} \\ \hline t\theta = \text{parent}(\text{homer}, \text{maggie}) \end{array}$$

- **Instanz:** Resultat einer Ersetzung

## Beispiel für Anfragen

parent(Y,X)?

*Y = grandpa*

*X = homer ;*

*Y = homer*

*X = bart ;*

...

*Y = marge*

*X = maggie*

parent(X,X)?

*no*

Variablen in Fakten möglich: *Allgemeingültige Fakten*

## Beispiel Fakt

```
hates(X, mr_burns) .
```

## Beispiel Anfrage

```
hates(maggie, mr_burns)?
```

*yes*

```
hates(foobar, mr_burns)?
```

*yes*

# Verknüpfte Anfragen

- Verknüpfung (*Konjunktion*) von Anfragen:
- Komma als logisches „und“:  
wahr, wenn alle Teilanfragen wahr sind

## Beispiel

```
parent(homer,bart), female(bart)?
```

*no*

```
parent(homer,bart), parent(marge,lisa)?
```

*yes*

# Gemeinsam benutzte Variablen

- Geltungsbereich: Gesamte verknüpfte Anfrage
- X: gleicher Wert in beiden Anfragen

## Beispiel

```
parent(homer,X), female(X)?
```

```
X = lisa ;
```

```
X = maggie
```

- **Regeln** oder Sätze:  
Zusammenfassen von Anfragen zu einer neuen Relation

## Definition

$A \leftarrow B_1, B_2, \dots, B_n.$

- A: Kopf des Satzes (neues Prädikat)
- $B_1, \dots, B_n$ : Rumpf des Satzes (eh. Anfragen)
- **Fakten**: Spezialfall für  $n = 0$
- *Hornklauseln*

## Beispiel

```
father(X,Y) ← parent(X,Y), male(X).
```

- Logisch oder prozedural lesbar
- Wiederverwendung möglich:

## Beispiel

```
grandfather(X,Y) ← father(X,Z), parent(Z,Y).
```

# Rekursion: Listen

- Benutzen des Satzkopfes im Rumpf möglich: **Rekursion**
- Beispiel Listen:  
Leere Liste: []  
Listenpaar: .(Head,Tail) oder [Head|Tail]

## Definition

`list([]).`

`list([H|T]) ← list(T).`

- z.B. Liste von Zahlen 1, 2 und 3:  
[1|[2|[3|[]]]] oder .(1,.(2,.(3,[])))
- Kurzform: [1,2,3]

- Aneinanderhängen von Listen:

```
append([], XL, XL).  
append([Y|YL], XL, [Y|ZL]) ← append(YL,XL,ZL).
```

- Relation zwischen 3 Listen
- Deshalb vielseitig einsetzbar:

```
member(X,YL) ← append(AL, [X|XL], YL).
```

- Meta-Prädikat: `not(X)`
- **Keine** echte logische Negation
- Wahr, wenn `X` keine Folge des Programms

## Beispiel

```
mother(X,Y) ← parent(X,Y), not(male(X))
```

- Ausführung: Ableiten der Anfrage  $t$  aus dem Programm  $P$
- Vorgehen:  
Passenden Regelkopf finden, mit dem Rumpf weitermachen
- Problem: Welcher Regelkopf passt auf die Anfrage?

- Methode zur Vereinheitlichung von Ausdrücken
- Unifikationsproblem:  $s \stackrel{?}{=} t$
- Ersetzung  $\theta$  (**Unifikator**) finden, so dass  $s\theta = t\theta$

## Beispiel

$\text{append}([1,2,3], V, X) \stackrel{?}{=} \text{append}([A|U], V, [A|W])$

---

**Unifikator:**  $\{A \mapsto 1, U \mapsto [2,3], X \mapsto [1|W]\}$

**Instanz:**  $\text{append}([1,2,3], [4,5], [1,W])$

# Allgemeinster Unifikator

- Oft viele Unifikatoren vorhanden
- **MGU**: *Most General Unifier*  
(auch: Kleinster gemeinsamer Unifikator)
- Andere Unifikatoren: Instanzen eines MGU

## Beispiel

$\text{male}(X) =? Y$

---

Ein Unifikator:  $\theta = \{Y \mapsto \text{male}(Z), X \mapsto Z\}$

Ein MGU:  $\sigma = \{Y \mapsto \text{male}(X)\}$

---

$\theta = \sigma\{Z \mapsto X\}$

Unifikationsproblem:  $S = \{s_1 =? t_1, \dots, s_n =? t_n\}$

- **Löschen** von überflüssigen Gleichungen
- **Aufspalten** von gleichen Prädikaten
- **Wenden** von Konstanten nach rechts
- **Anwenden** von Substitution

Anwenden der 4 Regeln, bis  $S = \{x_1 =? t_1, \dots, x_n =? t_n\}$   
( $x_i$  paarweise verschieden und nicht in  $t_i$  enthalten)

# Beispiel für Unifikation

## Beispiel

$\text{append}([1,2,3], V, X) =? \text{append}([A|U], V, [A|W])$

---

$\mapsto_2 \{ [1,2,3] =? [A|U], [4|5] =? V, X =? [A|W] \}$

$\mapsto_2 \{ 1 =? A, [2,3] =? U, V =? V, X =? [A|W] \}$

$\mapsto_3 \{ A =? 1, [2,3] =? U, V =? V, X =? [A|W] \}$

$\mapsto_4 \{ A =? 1, [2,3] =? U, V =? V, X =? [1|W] \}$

$\mapsto_3 \{ A =? 1, U =? [2,3], V =? V, X =? [1|W] \}$

$\mapsto_1 \{ A =? 1, U =? [2,3], X =? [1|W] \}$

---

$\text{append}([1,2,3], V, [1|W])$

- Unifikation **nicht** immer möglich:  
male(A) =? female(B)  
parent(X,Y,Z) =? parent(A,B)  
natural\_number(X) =? natural\_number(s(X))
- Weitere Tests möglich:
  - Clash
  - Occurs-Check

- Gegeben: Programm  $P$  und Anfrage  $q$  ( $= q_1, \dots, q_n$ )
- Gesucht: Aus  $P$  ableitbare Instanz von  $q$

## Vorgehen (vereinfacht)

Wähle einen Zielausdruck aus der Anfrage

Wähle eine Regel, deren Kopf auf den Zielausdruck passt

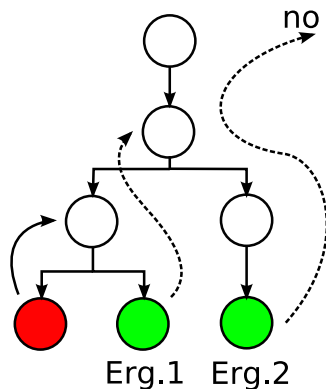
Ersetze den Zielausdruck durch den Rumpf dieser Regel

Ersetze Variablen in der ganzen Anfrage mittels Unifikator

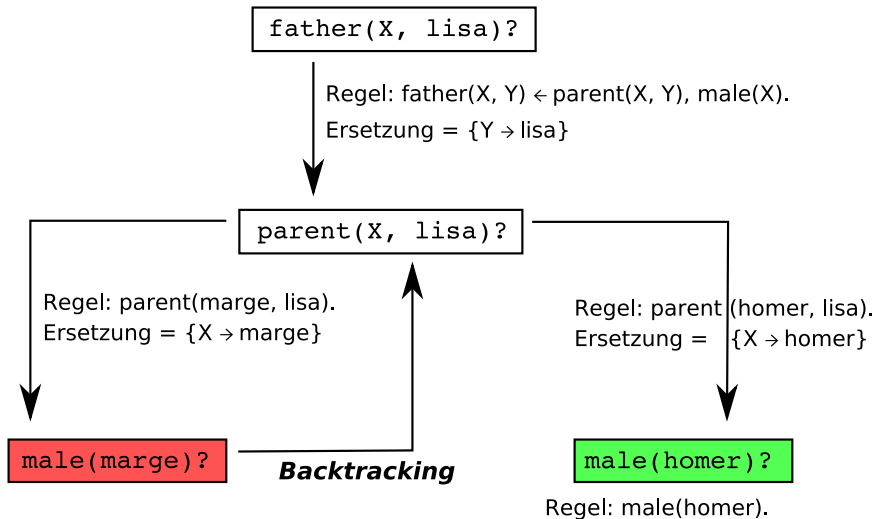
- Erfolgreich, wenn Anfrage komplett leer

# Backtracking

- Problem: Welche passende Regel anwenden?
- Lösung: Tiefensuche
- Bei Fehler: Zurück zur letzten Gabelung
- Bei Lösung: Ausgeben, evtl. ebenfalls zurück
- Begriff: **Backtracking**



# Backtracking



**Logisches Rätsel:** Reihe aus 4 Simpsons-Sammelkarten

- Platz 4 soll ein Elternteil von Platz 3 sein
- Platz 2 und Platz 4 sollen das gleiche Geschlecht haben
- Platz 1 soll weiblich sein
- Hinter Lisa soll eine männliche Person sein, aber nicht Bart
- Auf Platz 2 soll nicht Grandpa sein
- Keine Karte soll doppelt vorkommen

# Programm: Lösung - Teil 1

```
1 same_gender(X,Y) :- female(X), female(Y).
2 same_gender(X,Y) :- male(X), male(Y).
3
4 follow(A,B,[A,B|_]).
5 follow(A,B,[_|T]) :- follow(A,B,T).
6
7 unique([H|T]) :- not(member(H,T)), unique(T).
8 unique([]).
```





## Programm: Lösung - Teil 2

```
9 solve(X) :-
10     X = [A,B,C,D],
11     parent(D,C),
12     same_gender(D,B),
13     female(A),
14     follow(lisa,Z,X),
15     male(Z),
16     Z \= bart,
17     B \= grandpa,
18     unique(X).
```

- Ausdrucksstarke und deklarative Programmierung
- Schnelle Entwicklung möglich
- Gute Beweisbarkeit
- Ohne Optimierung oft langsam

Viele Implementierungen vorhanden, z.B.:

- **SWI-Prolog** (LGPL): sehr umfangreich, gut zum Lernen  
<http://www.swi-prolog.org/>
- **YAP Prolog** (Artistic License): sehr schnell  
<http://www.ncc.up.pt/~vsc/Yap/>
- **GNU Prolog** (GPL): nativer Compiler enthalten  
<http://gnu-prolog.inria.fr/>
- **Ciao Prolog** (GPL/LGPL)  
<http://www.clip.dia.fi.upm.es/Software/Ciao/>

-  STERLING, LEON and EHUD SHAPIRO:  
*The Art of Prolog.*  
The MIT Press, Cambridge, London, second edition, 1994.
-  BAADER, FRANZ and TOBIAS NIPKOW:  
*Term Rewriting and All That.*  
Cambridge University Press, Cambridge, 1999.
-  MERRITT, DENNIS: *Adventure in Prolog.*  
Amzi! inc., Lebanon, 2004.  
<http://www.amzi.com/AdventureInProlog/>.
-  WIKIPEDIA.  
*Logic programming, Prolog, ...*  
<http://www.wikipedia.org/>.