Prolog by example

Carsten Rösnick

Was ist Prolog?

- **Pro**grammation en **Log**ique (1970er)
- Nutzer erstellt **Datenbasis**
  - Definiert, was gilt in seinem Universum
  - Closed-world assumption
- Inferenz

Grundlagen (1)

- Ein erstes Beispiel:

```
v(a). v(b). v(c).                  Fakten (facts)
e(a,b). e(b,c).                    Fakten (facts)
t(X,Y) :- e(X,Z), e(Z,Y).          Regeln (goals)
```

- Konstantensymbole: `a,b,c`
- Funktions-/Relationssymbole: `v,e,t`
- Variablen: `X,Y`

Grundlagen (2)

```
t(X,Y) :- e(X,Z), e(Z,Y).
```
Regeln (goals)

- Konjunktion in Regeln: Komma “,”
- Disjunktion in Regeln: Semikolon “;”
- Fakt- und Regelende: Punkt “.”

Grundlagen (3): Quantoren

- $\forall x \forall y \exists z((Qxz \wedge Qzy) \vee Qyz \rightarrow Pxy)$
- Regel: `p(X,Y) :- (q(X,Z), q(Z,Y)); q(Y,Z).`
  - `Z` ist frei in p $\rightarrow$ `Z` Existenz-quantifiziert
  - `X,Y` gebunden in p $\rightarrow$ `X,Y` All-quantifiziert
- **Regeln mit gleicher Conclusio**

  = Disjunktion der Prämissen

```
t(X,Y) :- p(X), q(Y).
t(X,Y) :- r(X,Y).
```

$$\forall x \forall y((Px \wedge Qy) \vee Rxy \rightarrow Txy)$$

Ausführen von Prolog Programmen

- Unterstellen Nutzung von **GNU Prolog** [1]
- Fakten und Regeln in Textdatei **`data.pro`** sammeln
- Datenbasis **`data.pro`** in Prolog-Programm übersetzen mit **`gplc data.pro`**
- Programm ausführen: **`./data`**
- Anfragen stellen: weitere Lösungen mit **`;+Enter`**,
- alle Lösungen mit **`a+Enter`**

```
?- predicate(a,b,X).
X=c;
X=d
```

Beispiel: n! Berechnen (1)

- Vordefinierte Struktur: ganze Zahlen (arithmetische Ausdrücke)
- Erster (merkwürdiger) Versuch:

```
factorial(1) :- 1.
factorial(N) :- N*factorial(N-1).
```

- **Problem 1**: Merkwürdige Vermischung von Syntax, Semantik und Wahrheitswert der Interpretation.
- **Lösungsidee**:

```
factorial(0,1).
factorial(N,R) :- factorial(N-1,R1), R=N*R1.
```

Beispiel: n! Berechnen (2)

- **Problem 2**: Endlosrekursion
  - Grund (Auswertungsbaum) am Bsp. `factorial(2,R)`:

```
factorial(2,R).
  factorial(2-1,R).
    factorial(2-1-1,R).
      factorial(2-1-1-1,R).
        ...
```

- **Lösungsidee**:

```
factorial(N,R) :-
  N1 is N-1, factorial(N1,R1), R is N*R1.
```

Beispiel: n! Berechnen (3)

Merke: Auswertung arithmetischer Ausdrücke stets mit `Y is X`, auch um Endlosrekursion zu vermeiden

- **Problem 3: Endlosrekursion (cont.)**

```
factorial(2,R).
  N1 is 2-1.
  factorial(1,R).
    N1 is 1-1.
    factorial(0,R).
      N1 is 0-1.
      factorial(-1,R).
        ...
```

Beispiel: n! Berechnen (4)

- **Lösung** (final):

```
factorial(0,1).
factorial(N,R) :-
  N > 0,
  N1 is N-1,
  factorial(N1,R1),
  R is N*R1.
```

Warum löst das unser Problem?

- Und: Warum haben wir das “Problem” überhaupt?

  → Prolog's Suchstrategie: **Backtracking**

  Auch: Prolog **Auswertungsbaum**

- Erst **Regeln** in der **Reihenfolge ihres Auftretens** in der Datenbasis probieren
- Dann **Teilformeln** innerhalb einer Regel von **links nach rechts** versuchen zu verifizieren

Nochmal Rekursion (1)

- Auswertungsbaum Grund dafür, dass Reihenfolge relevant ist. Beispiel dafür:

```
parent(a,b). parent(b,c).
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- ancestor(X,B), parent(A,X).
```

- Zwei Lösungsideen. **Lösungsidee 1**:

  Menschen definieren:

```
human(a). human(b). human(c).
ancestor(A,B) :-
  human(X), ancestor(X,B), parent(A,X).
```

Nochmal Rekursion (2)

```
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- ancestor(X,B), parent(A,X).
```

```
ancestor(b,a).
   parent(b,a). fail
   ancestor(X1,a), parent(b,X1).
      parent(b,a). fail
      ancestor(X2,a), parent(X1,X2), parent(b,X1).
         parent(X1,a), parent(b,X1).
            ... fail
         ancestor(X3,a), parent(X2,X3), parent(X1,X2),
         parent(b,X1).
            ...
```

Nochmal Rekursion (3)

- **Lösung(sidee 2)**: tail recursion (good practice!)
- Rekursiv vorkommendes Prädikat stets als letzten Term der Konjunktion angeben
- Kurz: Rekursions**anfang** stets **vor** Rekursions**schritt**

```
ancestor(A,B) :- parent(A,X), ancestor(X,B).
```

→ spart hier sogar die eigenhändige Angabe der Menschen (`human(-)` Fakt)

not/1-Prädikat in GNU Prolog

- `not` ist kein vordefiniertes Prädikat
- Mögliche Definition: `not(X) :- \+ call(X).`
  - Vordefinierte Negation `\+`
  - Closed-world assumption
- Weitere Möglichkeit:

```
not_f(f(X)) :- f(X), !, fail.
```

Lösen von Rätseln: Türme von Hanoi

- Signatur der Regel:

```
move(
    Anzahl Scheiben,
    Startstab A,
    Zielstab B,
    Hilfsstab C).
```

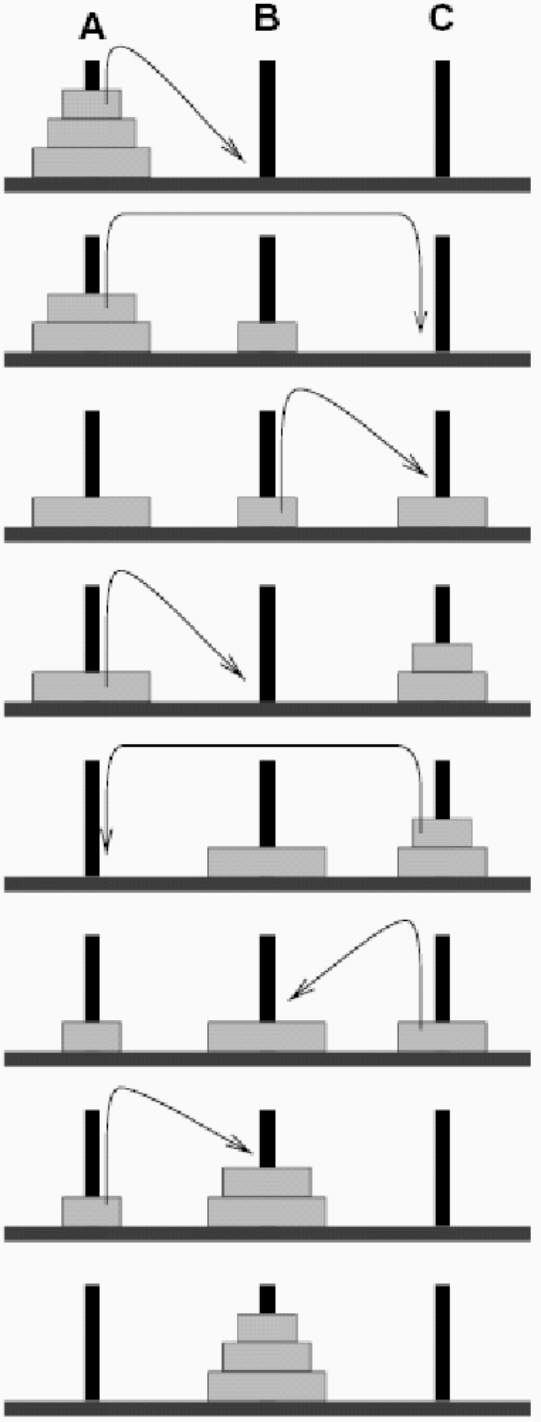

Abbildung: Lösungsstrategie Türme von Hanoi [4]

Ausblick

[...für kommendes Semester, nicht für FGdI :)]

- Cut-Operator (hier ist Vorsicht geboten!)
  - zur Auswertungssteuerung
- Listen
  - Auf Präfix einer Liste prüfen
  - Listenelemente Zählen (mit/ohne Duplikaten)
  - Listen umdrehen
- Ein-/Ausgabeoperationen
  - im Hanoi-Beispiel gesehen
- ...

Referenzen

[1] GNU Prolog Website: http://www.gprolog.org/

[2] Eine Einführung in Prolog und die logische Programmierung (.ppt), Mala Bachmann, 2001.

[3] http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

[4] http://www.mathematik.uni-ulm.de/sai/ss03/prog/Aufgaben/blatt04/