Kontrolle

Übersicht

- Alle Lösungen für ein Ziel erhalten
  - Semikolon, Failure-Driven Loop, findall/3
- Negation
- Disjunktion
- Suchbäume stutzen: Cut

Ziel

- Kennen dieser Programmiertechniken
- In der Lage sein, den Ablauf eines Prolog-Programms zu beeinflussen

Alle Lösungen eines Ziels

```
person(hans).
person(klara).
person(sabrina).
person(kevin).
```

Programm

```
?- person(X).
X = hans     ;
X = klara    ;
X = sabrina  ;
X = kevin    ;
no more solutions
```

Anfrage

Wie gehen wir vor, um alle Lösungen eines Ziels zu erhalten?

- wir kennen bereits: Eingabe eines Semikolons
- anstatt der manuellen Nachfrage möchte man jedoch einen *programmierbaren* Mechanismus haben

Failure-Driven Loop: Beispiel

Eine wichtige Programmiertechnik zum Bestimmen aller Lösungen ist der *Failure-Driven Loop*.

- die einzelnen Lösungen werden *nacheinander* verarbeitet
- typischerweise für Ausgabe auf Bildschirm oder Schreiben in Datei

```
person(hans).
person(klara).
person(sabrina).
person(kevin).

alle :-
  person(X),
  write(X), nl,
  fail.
```

Programm

```
?- alle.
no
```

Anfrage

```
hans
klara
sabrina
kevin
```

Ausgabe

Failure-Driven Loop: Verbesserung

Allerdings ist das Fehlschlagen der Anfrage unschön.

- Im Beispiel: Zweite Klausel für »alle«, die erst dann zum Zug kommt, wenn es keine weiteren Lösungen mehr gibt

```
person(hans).
person(klara).
person(sabrina).
person(kevin).

alle :-
  person(X),
  write(X), nl,
  fail.
alle.
```

Programm

```
?- alle.
yes
```

Anfrage

```
hans
klara
sabrina
kevin
```

Ausgabe

Failure-Driven Loop: Allgemein

Allgemeines Vorgehen beim *Failure-Driven Loop*:

- Sei `p(A, B, …)` ein Prädikat, von dem alle Lösungen bestimmt werden sollen
- Definiere Prädikat mit beliebigem Namen:
  - Klausel 1 besteht aus drei Teilen:
    - Aufruf von `p(A, B, …)` — danach sind `A, B, …` an Werte gebunden
    - Verarbeiten von `A, B, …` — typischerweise Ausgabe auf Bildschirm
    - `fail`
  - Klausel 2 ist ein Fakt.

```
bla :-
  p(A),
  write(A), nl,
  fail.
bla.
```

findall/3

```
person(hans).
person(klara).
person(sabrina).
person(kevin).
```

Programm

```
?- findall(W, person(W), Alle).
```

Alle = [hans, klara, sabrina, kevin]

Anfrage

In Prolog bereits eingebaut ist das Prädikat findall/3

- bestimmt alle Lösungen eines Ziels, liefert diese als Liste zurück
- sehr wichtig, wenn die Lösungsmenge als Ganzes weiterverarbeitet werden soll

findall/3

```
?- findall(x(P,M,V),
           (person(P), mutter(P,M), vater(P,V)),
           Ergebnis).
```

Ergebnis = [x(kevin, klara, hans), x(klara, kunigunde, gottfried)]

findall(+Term, +Ziel, –Liste)

- *Term* — wird für jede Lösung von *Ziel* zu *Liste* hinzugefügt
- *Ziel* — Ziel, das zu zeigen ist
- *Liste* — enthält für jede Lösung von *Ziel* die entsprechende Instanz von Term

Negation

Bisher konnten wir nur fragen, ob Prolog etwas bekannt ist (d.h. ob es etwas beweisen kann):

- Ist Klara eine Person? ?- person(klara).

Mit \+ wird gefragt, ob der nachfolgende Term *nicht* bewiesen werden kann:

- Ist Klara keine Person? ?- \+ person(klara).
- Gibt es niemanden, der eine Person ist? ?- \+ person(Jemand).

```
person(hans).
person(klara).
person(sabrina).
person(kevin).
```

Negation: Prolog und Logik

```
female(X) :-
   \+ male(X).

male(paul).
```

Programm

```
?- female(mary).
yes
```

Anfrage

Beachte: `\+` ist nicht dasselbe wie die logische Negation.

- aus den Fakten und Regeln folgt nicht, dass Mary weiblich ist!
- Zustandekommen der Antwort von Prolog:
  - `\+ p` gelingt dann, wenn `p` nicht bewiesen werden kann
- Zusätzliche Information kann `female(mary)` falsch machen
  - Die Zahl der beweisbaren Sätze kann mit zusätzlicher Information abnehmen
    ⇒ nicht monoton

Disjunktion

```
kind(K, Mami) :-
  tochter(K, Mami).
kind(K, Papi) :-
  sohn(K, Papi).
```

```
kind(K, Elter) :-
  tochter(K, Elter) ;
  sohn(K, Elter).
```

Rechte Seite einer Regel

- durch Komma getrennt
  - Einzelbedingungen müssen alle zusammen erfüllt sein
  - entspricht logischem »Und« (Konjunktion)
- durch Strichpunkt getrennt
  - es reicht, wenn eine der Einzelbedingungen erfüllt ist
  - entspricht logischem »Oder« (Disjunktion)

Suchbäume stutzen

```
max1(X, Y, X) :-
   X >= Y.

max1(X, Y, Y) :-
   X < Y.
```

Was geschieht bei der Anfrage max1(3, 2, Max)?

- Zwei Klauseln ⇒ Versuch mit erster Klausel, die auch gleich gelingt
- Prolog muss sich trotzdem den Entscheidungspunkt merken, da später evtl. Backtracking ausgelöst werden könnte (z.B. mit ;)

Suchbäume stutzen: Cut

Prolog weiss nicht, dass

- wenn die erste Klausel gelingt, ist es niemals möglich, dass die zweite gelingen kann
- anders gesagt: dass max1/3 *deterministisch* ist

Entscheidungspunkte kosten Zeit und Speicherplatz

- der Programmierer/die Programmiererin kann daher mit dem *Cut* — geschrieben als ! — den Suchbaum stutzen *(to prune)*
- ⇒ weniger Entscheidungspunkte
- ⇒ effizientere Programme

Suchbäume stutzen: Cut

```
max2(X, Y, X) :-
   X >= Y, !.

max1(X, Y, Y) :-
   X < Y, !.
```

Wirkung des Cut

- Wegschneiden aller Klauseln *unter* jener, die den Cut enthält
- Wegschneiden aller alternativen Lösungen für Konjunktionen in derselben Klausel *links* vom Cut

⇒ **weniger Entscheidungspunkte**

⇒ **effizientere Programme**

»Grüne« vs. »rote« Cuts

Der Cut sollte nur benutzt werden, um Teile des Suchbaums wegzuschneiden, die keine Lösungen enthalten

- sonst werden diese Lösungen nicht gefunden!

Bezeichnungen

- Grüne Cuts
  - schneiden überflüssige Suchbäume weg, die garantiert nichts zur Lösung beitragen
  - das Programm läuft effizienter, aber kommt zu denselben Lösungen
- Rote Cuts
  - schneiden Suchbäume weg, die eine Lösung enthalten würden
  - verändern die Lösungsmenge — die Bedeutung des Programms wird anders
  - meistens Programmierfehler

Falscher Einsatz des Cut

```
max_falsch(X, Y, X) :-
  X >= Y, !.

max_falsch(X, Y, Y).
```

Programm mit »rotem« Cut

```
?- max_falsch(5,2,M).
M = 5                ;
no more solutions
```

korrekt beantwortete Anfrage

```
?- max_falsch(5,2,2).
yes
```

falsch beantwortete Anfrage

Wirkung des Cut

Der Cut gelingt immer

- aber als »Seiteneffekt« wird der Suchbaum gestutzt
  - Wegschneiden aller Klauseln *unter* jener, die den Cut enthält
  - Wegschneiden aller alternativen Lösungen für Konjunktionen in derselben Klausel *links* vom Cut

```
x :- p.
x :- s.

p :- q, !.
p :- r.

q :- s.
q :- t.

s.
```

Programm