

Fehlersuche mit Prolog

Übersicht

- ◆ Auflisten der Wissensbasis
- ◆ Kästchenmodell (*Byrds Box*)
 - ▶ Komplexe Ziele: Nebeneinander kleben
 - ▶ Unterziele: Ineinander schachteln
- ◆ Trace-Modus
- ◆ Debug-Modus
 - ▶ Spy-Punkte verwalten
- ◆ Ausnahmen (*Exceptions*)
- ◆ Fehlermeldungen

Debuggen - 1

listing/0 und listing/1

Anzeigen, was Prolog beim Interpretieren eines Programmes verstanden hat.

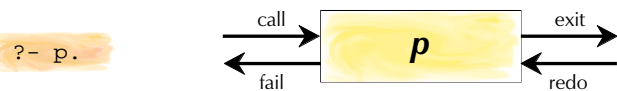
- ◆ Das Prädikat `listing` ist nützlich, wenn man wissen will, mit welcher Wissensbasis Prolog eigentlich beweist.
- ◆ Mit `listing/0` werden im Normalfall alle benutzerdefinierten Prädikate ausgegeben.
- ◆ Mit `listing/1` werden die Klauseln des Prädikat ausgegeben, das als Argument spezifiziert wurde

```
?- listing.
```

```
?- listing(q/0).
```

Debuggen - 2

Kästchenmodell



Ein einfaches Ziel *p* kann als Kästchen mit vier *Ports* (Ein- und Ausgänge) dargestellt werden.

- ◆ Zwei Eingänge
 - `call` — *p* soll zum ersten Mal bewiesen werden
 - `redo` — *p* soll über Backtracking ein weiteres Mal bewiesen werden
- ◆ Zwei Ausgänge
 - `exit` — *p* konnte bewiesen werden
 - `fail` — *p* konnte nicht bewiesen werden

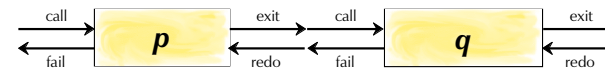
Debuggen - 3

Kästchenmodell: Konjunktion

```
?- p, q.
```

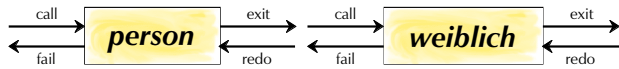
Konjunktiv verknüpfte Ziele ergeben nebeneinander verhängte Kästchen.

- ◆ Der Beweis beginnt mit dem ersten `call` (ganz links).
- ◆ Die Beweis gelingt mit dem letzten `exit` (ganz rechts).
- ◆ Mittleres `exit` wird mit `call` verbunden
- ◆ Mittleres `fail` wird mit `redo` verbunden



Debuggen - 4

Ein Beispiel



?- person(Wer), weiblich(Wer).

- call: person(Wer)
- exit: person(hans)
- call: weiblich(hans)
- fail: weiblich(hans)
- redo: person(hans)
- exit: person(klara)
- call: weiblich(klara)
- exit: weiblich(klara)

Wer = klara

```

person(hans).
person(klara).
person(gabi).
person(kevin).

weiblich(klara).
weiblich(gabi).
    
```

Wissensbasis

Debuggen - 5

trace/0 und notrace/0

Prolog kann dies selbst als **Tracing** ausgeben.

- ◆ Einschalten mit trace; Ausschalten mit notrace
- ◆ Nützlich beim Suchen von Programmierfehlern
 - ◆ Leider momentan kleine Unterschiede zwischen MacOS 3.6 und Vers. 3.8.4
 - ◆ Stationen zwischen exit und nächstem Entscheidungspunkt werden unterschlagen!

```

| ?- trace.
{The debugger will first creep -- showing everything (trace)}
yes
{trace}
| ?- person(Wer), weiblich(Wer).
1 1 Call: person(_187)?
1 1 Exit: person(hans)?
2 1 Call: weiblich(hans)?
2 1 Fail: weiblich(hans)?
1 1 Redo: person(hans)?
1 1 Exit: person(klara)?
2 1 Call: weiblich(klara)?
2 1 Exit: weiblich(klara)?
    
```

Wer = klara ? ;

1 1 Redo: person(klara) ?
 1 1 Exit: person(gabi) ?
 2 1 Call: weiblich(gabi) ?
 2 1 Exit: weiblich(gabi) ?

Wer = gabi ? ;

1 1 Redo: person(gabi) ?
 1 1 Exit: person(kevin) ?
 2 1 Call: weiblich(kevin) ?
 2 1 Fail: weiblich(kevin) ?

no

Strichpunkt erzwingt Backtracking

Debuggen - 6

Kästchenmodell: Verschachtelung

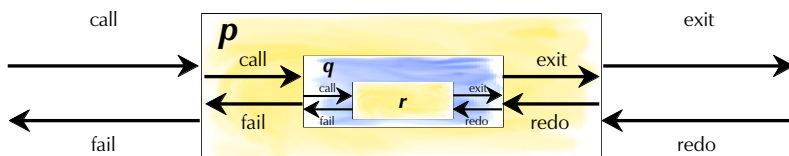
Unterziele, die bei Regeln durch Ersetzen des Rumpfs entstehen:

- ◆ Verschachtelung der Kästchen
- ◆ Das Ursprungsziel gelingt mit dem äussersten exit.

```

p :- q.
q :- r.
r.
    
```

?- p.



Debuggen - 7

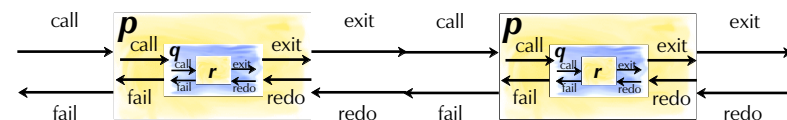
Verschachtelte Konjunktion...

Verschachteln und Hintereinanderstellen kombiniert

?- p, p.

```

?- p, p.
1 1 Call: p ?           4 1 Call: p ?
2 2 Call: q ?           5 2 Call: q ?
3 3 Call: r ?           6 3 Call: r ?
3 3 Exit: r ?           6 3 Exit: r ?
2 2 Exit: q ?           5 2 Exit: q ?
1 1 Exit: p ?           4 1 Exit: p ?
    
```



Debuggen - 8

Zahlendeutung

Was bedeuten die Zahlen vor den Ports?

4 1 Call: p ?
5 2 Call: q ?

1. Zahl

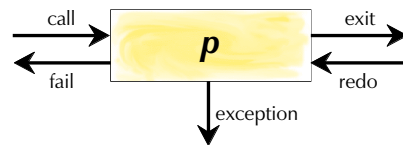
- Bei Call-Ports: Anzahl durchschrittener Call-Eingänge, seit Beginn der Anfrage
- Bei andern Ports: Bezug auf den entsprechenden Call-Port
 - Nummern identifizieren Aufrufe (für Version < 3.7: Kästchen) eindeutig!

2. Zahl

- Verschachtelungstiefe beim Beweisen

Fragezeichen vor 1. Zahl bedeutet Entscheidungspunkt

Die Ausnahme: Ein Notausgang



Notausgang für Ausnahmefälle (exceptions)

- Mit den 4 Ports muss jede Anfrage bewiesen ("yes") werden oder scheitern ("no").
- Der Exception-Ausgang erlaubt es, quer zum Beweisvorgang aus Kästchen herauszukommen. (Weder "no" noch "yes" am Schluss)
- Exceptions wandern gegen Aussen, und müssen immer durch den Exception-Port (ausser sie werden explizit aufgefangen)

debug/0, nodebug/0, spy/1, nospy/1

Manchmal ist es mühsam, alle Prädikate zu tracen...

- Debug-Modus: Einschalten mit `debug`, Ausschalten mit `nodebug`
- Prolog zeigt zunächst nur den Trace von Prädikaten, auf die mit `spy/1` ein `spy`-Punkt gesetzt wurde (+)
 - Mit `RET` oder `c` (*creep*) kriecht man wie beim `trace`-Modus weiter
 - Mit `l` (*leap*) springt man zum nächsten Port eines Prädikats mit `spy`-Punkt
 - Mit `n` (*nodebug*) wird die Anfrage ohne Tracing beendet
 - Mit `h` (*help*) gibt's eine Menuübersicht
- Löschen eines `spy`-Punkts mit `nospy/1`

```
?- debug.
{The debugger will first leap -
 - showing spyoints (debug)}
yes
{debug}
| ?- spy(q/0).
{Spypoint placed on user:q/0}
yes
{debug}
| ?- p, P.
+ 2 2 Call: q ?
  3 3 Call: r ?
  3 3 Exit: r ? 1
+ 2 2 Exit: q ? 1
+ 5 2 Call: q ? n
yes
{debug}
```

Ausnahmefälle für Fehler

Fehlermeldung durch Ausnahmen

- Moderne Prologs, die sich am ISO-Standard ausrichten, melden Fehler durch *exceptions*.
- Es werden dabei unterschiedliche Klassen von Fehler unterschieden
 - Existenzfehler** (*existence error*): Aufgerufenes Prädikat existiert nicht
 - Syntaxfehler** (*syntax error*): Irgendetwas im Programmtext ist syntaktisch falsch
 - Instantiierungsfehler** (*instantiation error*): Bei einer Anfrage war ein Argument ungenügend instantiiert
 - Typenfehler** (*type error*): Beim Beweisen war ein Argument vom falschen Typ.
 - Systemfehler** (*system error*): Es ist ein Systemfehler aufgetreten.

```
{EXISTENCE ERROR: t: procedure user:t/0 does not exist}
```

- Um Fehler zu beheben, muss man die Fehlermeldung verstehen!