Fehlersuche mit Prolog

Übersicht

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

listing/0 und listing/1

Anzeigen, was Prolog beim Konsultieren eines Programmes verstanden hat.

- Das Prädikat `listing` ist nützlich, wenn man wissen will, mit welcher Wissensbasis Prolog eigentlich beweist.
- `listing/0` gibt alle benutzerdefinierten Prädikate aus.

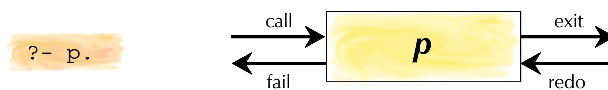
```
?- listing.
```

- `listing/1` gibt nur dasjenige Prädikat aus, das als Argument in der Prädikatsfunktor/Stelligkeit-Notation spezifiziert wurde

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


Kästchenmodell

```
?- p.
```



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

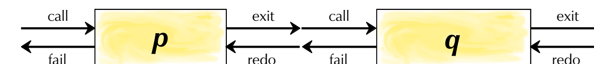
- Zwei Eingänge
  - *call* — *p* soll zum ersten Mal bewiesen werden
  - *redo* — *p* soll über Backtracking ein weiteres Mal bewiesen werden
- Zwei Ausgänge
  - *exit* — *p* konnte bewiesen werden
  - *fail* — *p* konnte nicht bewiesen werden

Kästchenmodell: Konjunktion

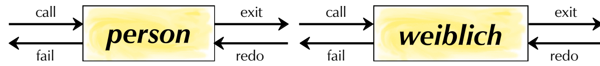
```
?- p, q.
```

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



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

trace/0 und notrace/0

Prolog kann dies selbst als *Tracing* ausgeben.

- Einschalten mit `trace`; Ausschalten mit `notrace`
- Nützlich beim Suchen von Programmierfehlern
  - Achtung: Beim Backtracking werden Stationen zwischen exit und nächstem Entscheidungspunkt 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) ?
   3  1  Call: weiblich(klara) ?
   3  1  Exit: weiblich(klara) ?
```

```
Wer = klara ? ;
   1  1  Redo: person(klara) ?
?  1  1  Exit: person(gabi) ?
   4  1  Call: weiblich(gabi) ?
   4  1  Exit: weiblich(gabi) ?
```

```
Wer = gabi ? ;
   1  1  Redo: person(gabi) ?
   1  1  Exit: person(kevin) ?
   5  1  Call: weiblich(kevin) ?
   5  1  Fail: weiblich(kevin) ?
no
```

Strichpunkt erzwingt Backtracking

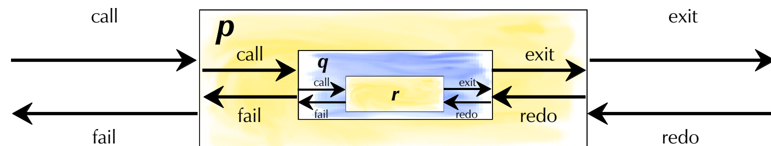
Kästchenmodell: Verschachtelung

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.
```

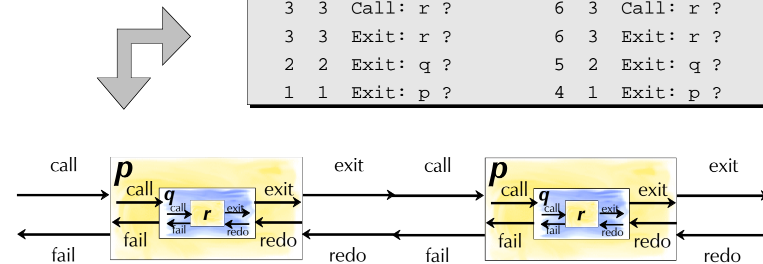


Verschachtelte Konjunktion…

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 ?
```



Zahlendeutung

Was bedeuten die Zahlen vor den *Ports*?

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

1. Zahl

- Bei *Call-Ports*: Zählt die durchschrittenen *Call*-Eingänge, ab Beginn der Anfrage
- Bei andern Ports: Bezug auf den entsprechenden *Call-Port*
  - Jede Nummer identifiziert eindeutig einen Aufruf!

2. Zahl

- Verschachtelungstiefe beim Beweise

Fragezeichen vor 1. Zahl bedeutet Entscheidungpunkt

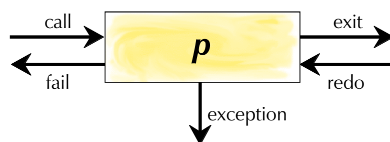
debug/0, nodebug/0, spy/1, nospy/1

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 fertig bewiesen
  - Mit a (abort) wird die Anfrage abgebrochen
  - Mit h (*help*) gibt's eine Menüübersicht
- Löschen eines *spy*-Punkts mit `nospy/1`

```
| ?- debug.
{The debugger will first leap -
- showing spypoints (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 ? l
+ 2  2  Exit: q ? l
+ 5  2  Call: q ? n
yes
{debug}
```


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)

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!