Einführung in das Programmieren – Prolog
Sommersemester 2006

Teil 5: Programmfluß

Version 1.0

Gliederung der LV

Teil 1: Ein motivierendes Beispiel

Teil 2: Einführung und Grundkonzepte
- Syntax, Regeln, Unifikation, Abarbeitung

Teil 3: Arithmetik

Teil 4: Rekursion und Listen

Teil 5: Programmfluß
- Negation, Cut

Teil 6: Verschiedenes
- Ein-/Ausgabe, Programmierstil

Teil 7: Wissensbasis
- Löschen und Hinzufügen von Klauseln

Teil 8: Fortgeschrittene Techniken
- Metainterpreter, iterative Deepening, PTTP, Differenzlisten, doppelt verkettete Listen

Deklarative vs. prozedurale Semantik

Deklarative Semantik

- Folgt eine Aussage (ein Ziel) logisch aus dem Wissen, das im Programm repräsentiert wird?
- deklarative Semantik liefert kein Verfahren, wie der gesuchte Beweis zu finden ist.

$\longrightarrow$ keine Vorgaben über Reihenfolge/Struktur der Abarbeitung usw.

Prozedurale Semantik

- Ein Verfahren (Algorithmus) zur Erlangung eines Beweis bestimmt die Semantik

$\longrightarrow$ Reihenfolge/Struktur der Abarbeitung ist festgelegt

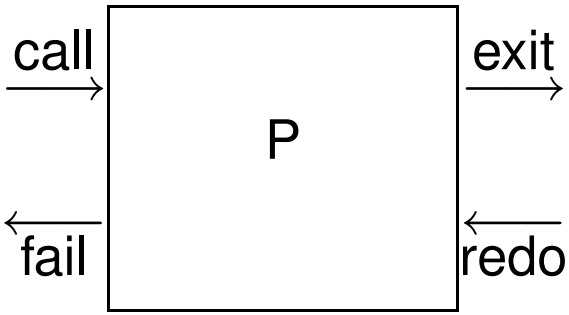
- Dieses Verfahren wird von Prologsystemen realisiert und bestimmt deren Reaktion

Einfluß der Klauselreihenfolge

```
wechsel(p1, p2).
wechsel(X, Y) :- wechsel(Y, X).
```

- Was passiert bei der Anfrage `?- wechsel(A, B).` ?
- Was passiert bei Vertauschung der Reihenfolge der Klauseln im Programm?

Boxmodell



call	:	P wird zum ersten Mal aufgerufen.
exit	:	P war erfolgreich.
fail	:	P schlug fehl.
redo	:	P wird durch Backtracking aufgerufen.

- Boxmodell ist ein Modell der Abarbeitung eines Prologprogramms
- Unterstützt prozedurale Sichtweise
- Tracing entspricht dem Kontrollfluß durch die Boxen.
- Jedem zu beweisenden Teilziel entspricht eine Box
- Ein Ziel, das mehrmals mit “exit” verlassen werden kann heisst *backtrackfähig*.

Standardprädikate zur Steuerung des Programmablaufs

true gelingt immer

fail scheitert immer

! Cut

not(P) gelingt genau dann, wenn P scheitert

P1;P2 Disjunktion der Ziele P1 und P2

P1,P2 Konjunktion der Ziele P1 und P2

P1,P2 Konjunktion der Ziele P1 und P2

repeat true, liefert beliebig viele Choicepoints

Beispiel für fail

```
drucke_alle_elemente(L):-
        member(X,L),
        write(X),
        nl,
        fail.
drucke_alle_elemente.
```

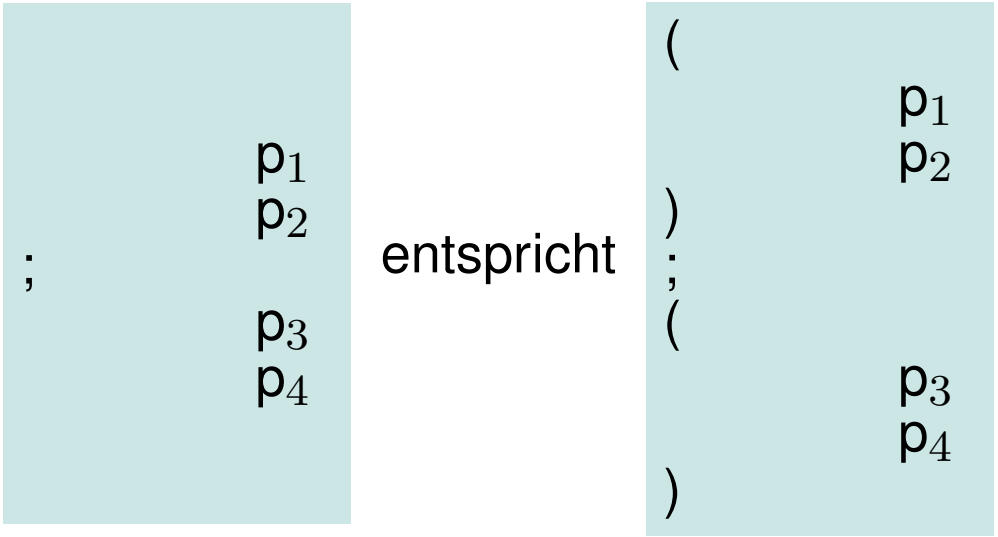
```
?- drucke_alle_elemente([a,b,c]).

a
b
c

Yes
```


Oder

- ; stellt Disjunktion (oder) dar
- Wie ist die Klammerung?



repeat

- repeat gelingt immer, liefert unendlich viele Choicepoints.
  Definiert als

```
repeat.
repeat :- repeat.
```

Beispiel: `poll` wartet solange, bis ein Zeichen im Puffer ist und verarbeitet es dann.

```
poll:-
        repeat,
        sleep(1),
        es_ist_ein_zeichen_im_puffer,
        verarbeite_zeichen.
```

Negation als Fehlschlag

Prinzip der Weltabgeschlossenheit

- Es wird angenommen, daß Aussagen nur dann wahr sind, wenn sie aus Aussagen des Prologprogramms folgen.
- Kann eine Aussage nicht bewiesen werden, wird sie als falsch angenommen (bzw. die Negation der Aussage als wahr).
- `Yes`: Bedeutet, daß Aussage bewiesenermaßen wahr ist.
- `No`: Bedeutet, daß Aussage nicht bewiesenermaßen wahr ist.

Der not-Operator (\+)

- `\+` ist alternative Notation für `not`
- `not Goal` hat Erfolg, falls `Goal` fehlschlägt.

```
?- not(element(17, [1, 2, 5])).
Yes
```

  gleiche Semantik: `?- \+ element(17, [1, 2, 5]).`
- ***Negation als Fehlschlag***: Prolog's Negation ist definiert als Fehlschlag, einen Beweis zu finden.
- Bei `\+ Goal` werden keine Variablenbindungen vorgenommen.
- `\+` nicht anwendbar auf Fakten oder Kopf einer Regel.

Der not-Operator (\+)

- `\+\+ Goal` testet, ob `Goal` erfüllt werden kann; nimmt jedoch keine Variablenbindungen vor.

```
?- f(X,g(Y)) = f(h(i),g(i)).

X = h(i)
Y = i

?- \+ \+ f(X,g(Y)) = f(h(i),g(i)).

X = _G320
Y = _G318 ;
```

Beispiel

```
verheiratet(peter, erika).
verheiratet(paul, susanne).
verheiratet(erich, carla).
verheiratet(harald, elke).

single(Person) :-
        \+ verheiratet(Person, _),
        \+ verheiratet(_, Person).
```

```
?- single(carla).
No.
?- single(claudia).
Yes
```

Probleme mit Backtracking

- Beispiel: Entfernen von “Doppelgängern” in einer Liste

```
remove_duplicates([ ], [ ]).

remove_duplicates([Kopf | Rumpf], Ergebnis) :-
        member( Kopf, Rumpf),
        remove_duplicates( Rumpf, Ergebnis).

remove_duplicates([Kopf| Rumpf], [Kopf|Ergebnis]) :-
        remove_duplicates( Rumpf, Ergebnis).
```

- diese Definition führt zu alternativen Lösungen, die falsch sind:

```
?- remove_duplicates( [a, b, b, c, a], Liste).
Liste = [b, c, a];
Liste = [b, b, c, a];
Liste = [a, b, c, a];
Liste = [a, b, b, c, a]
Yes
```

Der Cut Operator

- Manchmal soll verhindert werden, daß Prolog bei bestimmten Auswahlpunkten Backtracking durchführt, das eigentlich möglich wäre
- Gründe hierfür sind
  - Verhindern falscher Lösungen
  - Effizienzgründe

Notation in Prolog: !

Wirkungsweise des Cut

- Klausel C:

```
A :- B1, ..., Bk, !, Bk+1, ..., Bn.
```

Angenommen, B1, ..., Bk sind bereits bewiesen.

Nun werden Bk+1, ... Bn bewiesen.

Wenn alle bewiesen werden können $\longrightarrow$ Verhalten ganz normal

Wenn ein Bk+i nicht bewiesen werden kann, dann werden alternative Lösungsmöglichkeiten nur bis zum Cut durchsucht, d.h. für Bk+1, ..., Bn, jedoch nicht für B1, ..., Bk und auch nicht für weitere Regeln für A.

Wirkungsweise des Cut

Illustrierendes Programm (zunächst ohne Cut):

```
p1(1-1).                p1(1-2).

p2(2-1).                p2(2-2).

p3(3-1).                p3(3-2).

p4(4-1).                p4(4-2).

p:-
        p1(X1),
        p2(X2),
        p3(X3),
        p4(X4),
        write(versuch1(p1(X1), p2(X2), p3(X3), p4(X4))),nl,
        fail.

p:- write(versuch2),nl.
```

Wirkungsweise des Cut

Ergebnis:

```
versuch1(p1(1-1), p2(2-1), p3(3-1), p4(4-1))
versuch1(p1(1-1), p2(2-1), p3(3-1), p4(4-2))
versuch1(p1(1-1), p2(2-1), p3(3-2), p4(4-1))
versuch1(p1(1-1), p2(2-1), p3(3-2), p4(4-2))
versuch1(p1(1-1), p2(2-2), p3(3-1), p4(4-1))
versuch1(p1(1-1), p2(2-2), p3(3-1), p4(4-2))
versuch1(p1(1-1), p2(2-2), p3(3-2), p4(4-1))
versuch1(p1(1-1), p2(2-2), p3(3-2), p4(4-2))
versuch1(p1(1-2), p2(2-1), p3(3-1), p4(4-1))
versuch1(p1(1-2), p2(2-1), p3(3-1), p4(4-2))
versuch1(p1(1-2), p2(2-1), p3(3-2), p4(4-1))
versuch1(p1(1-2), p2(2-1), p3(3-2), p4(4-2))
versuch1(p1(1-2), p2(2-2), p3(3-1), p4(4-1))
versuch1(p1(1-2), p2(2-2), p3(3-1), p4(4-2))
versuch1(p1(1-2), p2(2-2), p3(3-2), p4(4-1))
versuch1(p1(1-2), p2(2-2), p3(3-2), p4(4-2))
versuch2

Yes
```

Wirkungsweise des Cut

Illustrierendes Programm (jetzt mit Cut zwischen p2 und p3):

```
p1(1-1).                p1(1-2).

p2(2-1).                p2(2-2).

p3(3-1).                p3(3-2).

p4(4-1).                p4(4-2).

p:-
        p1(X1),
        p2(X2),
        !,
        p3(X3),
        p4(X4),
        write(versuch1(p1(X1), p2(X2), p3(X3), p4(X4))),nl,
        fail.

p:- write(versuch2),nl.
```

Wirkungsweise des Cut

Ergebnis:

```
versuch1(p1(1-1), p2(2-1), p3(3-1), p4(4-1))
versuch1(p1(1-1), p2(2-1), p3(3-1), p4(4-2))
versuch1(p1(1-1), p2(2-1), p3(3-2), p4(4-1))
versuch1(p1(1-1), p2(2-1), p3(3-2), p4(4-2))

No
```

Nachdem also einmal festgelegt wurde, daß X1=1-1 und X2=2-1 sind, werden keine weiteren Alternativen für p1 und p2 betrachtet; und auch die zweite Klausel für p wird ignoriert.

Wirkungsweise des Cut

- Ein Cut schneidet alle Klauseln mit demselben Prädikat aus dem Suchraum, die unter der aktuellen Klausel mit dem Cut stehen.
- Ein Cut schneidet alle alternativen Lösungen der Ziele heraus aus dem Suchraum, die in der Klausel links vom Cut stehen. D.h. für Ziele links vom Cut wird höchstens eine Lösung betrachtet.
- Ein Cut beeinflußt nicht die Ziele zu seiner Rechten.

Probleme mit Backtracking

- Korrektes Programm zum Entfernen von Doppelgängern:

```
remove_duplicates([ ], [ ]).

remove_duplicates([Kopf| Rumpf], Ergebnis) :-
            member( Kopf, Rumpf),
            !,
            remove_duplicates( Rumpf, Ergebnis).

remove_duplicates([Kopf| Rumpf], [Kopf|Ergebnis]) :-
            remove_duplicates( Rumpf, Ergebnis).
```

Green and Red Cuts

Sprachliche Unterscheidung, keine syntaktische

Green Cuts

- Ein Cut, der die Bedeutung des Programms nicht verändert, aber einen Teil des Suchbaums abschneidet, in der garantiert keine Lösung des Problems liegen kann.

Red Cuts

- Ein Cut, der die Bedeutung des Programms (z.T. radikal) ändert.

Beispiel für Green und Red Cuts

- zwei Varianten von `merge(X,Y,Z)`, bei dem zwei geordnete Listen von Integern in eine einzelne sortierte Liste verbunden werden.

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
        X =< Y,
        merge([Xs], [Y|Ys], [Zs]).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
        X > Y,
        merge([X|Xs], Ys, Zs).

merge(Xs, [], Xs).

merge([], Ys, Ys).
```

Beispiel für Green Cuts

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
            X =< Y,
            !,
            merge([Xs], [Y|Ys], [Zs]).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
            X > Y,
            merge([X|Xs], Ys, Zs).

merge(Xs, [],Xs) :- !.

merge([],Ys,Ys).
```

$\rightsquigarrow$ Cuts machen Programm deterministischer durch Beschneiden unnützer Alternativen

Beispiel für Red Cuts

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
            X =< Y,
            !,
            merge([Xs], [Y|Ys], [Zs]).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
            merge([X|Xs], Ys, Zs).

merge(Xs, [],Xs) :- !.

merge([],Ys,Ys).
```

$\rightsquigarrow$ Cuts machen Programm schneller durch Vermeiden unnützer Tests

If-Then-Else Konstrukte

- Prädikat `if_then_else(P,Q,R)` ist wahr, falls `P` und `Q` wahr sind oder falls `not P` und `R` wahr sind.

```
if_then_else(P, Q, R) :- P, Q.

if_then_else(P, Q, R) :- not P, R.
```

- Falls P nicht gilt, dann wird zweimal versucht, P zu beweisen
  - Zeitverschwendung
  - Problem bei Seiteneffekten

    ```
    p :- write(...), fail
    ```

    Schreibt ... zweimal aus

Variante mit Cut:

```
if_then_else(P, Q, R) :-
                P,
                !,
                Q.

if_then_else(P, Q, R) :- R.
```

If-Then-Else Konstrukte

es gibt ein spezielles syntaktisches Konstrukt für if-then-else: ->

```
p:-
                q1,
                :
                ql
        ->
                r1,
                :
                rm
        ;
                s1,
                :
                sn
```

steht für

```
p:-
                q1,
                :
                ql,
                !,
                r1,
                :
                rm.
p:-
                s1,
                :
                sn.
```

Alternative Definitionen von element

```
element(E, [E | Rest]).

element(E, [Kopf | Rest]) :- element(E, Rest).
```

```
element1(E, [E | Rest]) :- !.

element1(E, [Kopf | Rest]) :- element(E, Rest).
```

```
element_check(E, [E | Rest]).

element_check(E, [Y | Rest]) :-
            E =\= Y,
            element_check(E, Rest).
```

Cut and Fail

Implementierung der Negation

- Negation als Fehlschlag wird wie folgt implementiert

```
not X :-
            X,
            !,
            fail.

not X.
```

Gleiche Variablen

- `same_var(X,Y)` testet, ob `X` und `Y` dieselbe Variable sind

```
same_var(abc,Y) :-
            var(Y),
            !,
            fail.

same_var(X,Y) :-
            var(X),
            var(Y).
```

Mengenprädikate

- Prolog liefert immer eine Antwort je Anfrage zurück
- Manchmal kann es aber sinnvoll sein, alle möglichen Antworten zu bekommen.
- Standardprädikate vorhanden, die solche Prädikate realisieren

findall

- `findall(+Term,+Ziel,-Liste)`
  1. Liste enthält alle Instanzen von Term, für die Ziel beweisbar ist.
  2. Für jeden erfolgreichen Beweis eine Instanz
  3. leere Liste, falls Ziel nicht erfüllbar
  4. Liste enthält u.U. identische Instanzen

```
?- findall(Tiere,ist_groesser_als(elefant,Tiere),
Tierliste).
Tierliste = [pferd, esel, hund, affe]
Yes
```

bagof und setof

- `bagof(+Term,+Ziel,-Liste)`
  1. Ähnlich wie `findall`, aber `bagof` ist backtrackfähig (bzgl. weiterer freier Variablen), `findall` nicht
  2. `bagof` failt, falls Ziel nicht erfüllen

- `setof(+Term,+Ziel,-Menge)`
  1. `Menge` enthält alle Instanzen von Term, für die das Ziel beweisbar ist.
  2. `Menge` enthält keine Duplikate