Einführung in das Programmieren – Prolog
Sommersemester 2006

Teil 2: Einführung und Grundkonzepte

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

Organisatorisches

- Vorlesung jeweils von 9:30 Uhr bis 13:00 Uhr
- Bearbeitung von Übungsaufgaben am Rechner in Raum S202/C005
  - Beginn ab ca. 13:30 Uhr, geplantes Ende ca. 17 Uhr
  - Arbeiten in Gruppen (2-3 Leute)
  - Betreuung durch mich und einen Mitarbeiter
    das heißt, Sie müssen sich gegenseitig helfen...
  - Forum
- Schein: gibt es nicht
  - Wer trotzdem einen braucht: bitte Anerkennung und Modalitäten mit dem jeweiligen Prüfungsamt klären und mir *rechtzeitig* Bescheid geben.

Unterlagen zur Vorlesung

Homepage

- http://www.ke.informatik.tu-darmstadt.de/lehre/ss06/prolog/

Weitere Informationen

- Webseite von SWI-Prolog `http://www.swi-prolog.org/`

Programmierparadigmen

Imperatives Programmieren

- Beschreibung des Algorithmus im Vordergrund (WIE)
  1. Problembeschreibung
  2. Lösungsweg
  3. Algorithmus
  4. Programm
  5. Lösungssuche
- Beispiele
  - Prozedurale Sprachen (Pascal, Modula, C, ..)
  - Objektorientierte Sprachen (Java, C++, Eifel, Smalltalk ..)

Programmierparadigmen

Funktionales Programmieren

- Funktionaler Zusammenhang des Problems wird beschrieben
- Beispiele: LISP, ML, Scheme, ..

Deklaratives Programmieren

- Formale Beschreibung der Ausgangssituation, formulieren der Problemstellung (WAS)
- Problemlösung übernimmt zugrundeliegendes System
  1. Problembeschreibung
  2. Angabe von Wissen über das Problem als Fakten und Regeln
  3. Lösungssuche
- Beispiele: Prolog und Varianten (Mercury, ..), Gödel

PROLOG: PROgramming in LOGic

Historische Entwicklung:

- Erste Ansätze Anfang der 70iger (Kowalski, Colmerauer)
- Warren Abstract Machine (WAM)
- Japanese Fifth Generation Project (80iger)
- Present: Constraint Logic Programming extensions.

Literaturhinweise

I. Bratko. *Prolog – Programmierung für Künstliche Intelligenz*. Addison-Wesley, 1987.

W. F. Clocksin und C. Mellish. *Programmieren in Prolog*. Springer, 1990.

R. Cordes, R. Kruse, H. Langendörfer und H. Rust. *Prolog – Eine methodische Einführung*. Vieweg, 1988.

N. E. Fuchs. *Kurs in Logischer Programmierung*. Springers Angewandte Informatik, 1990.

M. Hanus. *Problemlösen mit Prolog*. Teubner, 1987.

H. Kleine Büning. *Prolog – Grundlagen und Anwendungen*. Teubner, 1986.

R. A. O'Keefe. *The Craft of Prolog*. MIT Press, 1991.

L. Sterling und E. Shapiro. *Prolog – Fortgeschrittene Programmiertechniken*. Addison-Wesley, 1988.

Fakten

Umgangssprachlicher Fakt:

- “Ein Elefant ist größer als ein Pferd”

Darstellung in Prolog:

```
groesser(elefant, pferd).
```

Weitere Fakten:

```
groesser(pferd, esel).
groesser(esel, hund).
groesser(esel, affe).
```

⇝ ***Wissensrepräsentation als Fakten in Prolog-WB***

Anfragen

Anfragen an die Prolog-WB:
“ist ein Pferd größer als ein Esel?”

```
?- groesser(pferd, esel).
```
Yes

```
?- groesser(affe, esel).
```
No

```
?- groesser(pferd,X).
```
X = esel
Yes

Problem

Problem: Folgende Anfrage schlägt fehl

```
?- groesser(elefant, affe).
No
```

Wünschenswert wäre es aber schliessen zu können, dass diese Aussage wahr ist.

$\rightsquigarrow$ `groesser/2` allein ist nicht genug

$\rightsquigarrow$ Gewollt ist die transitive Hülle des Prädikats `groesser/2`, d.h. ein Prädikat, das immer zutrifft, falls vom ersten Tier zum zweiten das Prädikat in einer Kette abgeleitet werden kann.

Regeln

Die folgenden beiden Regeln leisten das Gewollte und definieren `ist_groesser_als` als die transitive Hülle von `groesser`

```
ist_groesser_als(X, Y) :-
        groesser(X, Y).

ist_groesser_als(X, Y) :-
        groesser(X, Z),
        ist_groesser_als(Z, Y).
```

Weiterer Vorteil: Bei Ergänzung der Wissensbasis werden ableitbare Fakten bei Bedarf “automatisch” ergänzt.

```
groesser(wal, elefant).
```

Regel: Generelle Aussage über Beziehung zwischen Objekten.

Regeln

Nach Hinzufügen der Regeln:

```
?- ist_groesser_als(elefant, affe).
Yes
```

Wir können auch Anfragen mit Variablen “X” machen:

```
?- ist_groesser_als(X, pferd).

X = elefant ;
X = wal ;
No
```

Eingabe von “`;`” führt zu alternativen Lösungen. `No` am Ende sagt, keine weitere Lösung gefunden.

Regeln

Weitere Beispielsanfragen:

```
?- ist_groesser_als(pferd, X).

X = esel;
X = hund;
X = affe;
No
```

Verknüpfung von Anfragen:

```
?- ist_groesser_als(esel,X), ist_groesser_als(X,
affe).
No
```

Beispiel: Familienbeziehungen

```
maennlich(paul).      maennlich(fritz).     maennlich(steffen).   maennlich(robert).
weiblich(karin).      weiblich(lisa).       weiblich(maria).      weiblich(sina).

vater(steffen, paul). vater(fritz, karin).  vater(steffen, lisa). vater(paul, maria).
mutter(karin, maria). mutter(sina, paul).

elternteil(E, Kind) :- vater(E, Kind).
elternteil(E, Kind) :- mutter(E, Kind).

bruder(X, Y) :-
        maennlich(X),
        elternteil(E, X),
        elternteil(E, Y),
        X \== Y.

vorfahre(X, Y) :- elternteil(X, Y).
vorfahre(X, Y) :-
        elternteil(X, Z),
        vorfahre(Z, Y).
```

Syntax von Prolog

Terme

- ***Terme*** können sein entweder Zahlen, Atome, Variablen oder Strukturen

Atome

- Folge von alphanumerischen Zeichen beginnend mit kleinem Buchstaben

  x, abcXYZ, x_123

- Folge von Zeichen eingeschlossen in ' und '

  'U76', 'Das ist ein Prologatom.', 'das auch'

- Sonderzeichen $(:, -, +, *, =, >, \&, ..)$ und *einige* Kombinationen davon

Achtung: * und *** sind Atome, a*b dagegen nicht (das ist ein Term)

Prolog-Prädikat zum Test auf Atom: atom/1

Variablen

- beginnen mit einem grossen Buchstaben oder dem *Unterstrich*

  X, Elephant, _XYZ, Variable

  Prolog-Prädikat zum Test auf Variablen: var/1

- Variablen beginnend mit _ sind ***anonyme Variablen***
  - implizite Kommentierung: man drückt damit aus, daß der Wert der Variable keine Rolle spielt
  - Es werden in der Antwort keine Lösungen für anonyme Variablen zurückgegeben
  - Compiler warnt, falls Variablen nur einmal in Regel auftauchen $\longrightarrow$ nicht für anonyme Variablen
  - Die anonyme Variable _ hat besondere Bedeutung: Jedes Auftreten

```
?- vater(X,X).
No
?- vater(_X,_X).
No
?- vater(_,_).
Yes
```

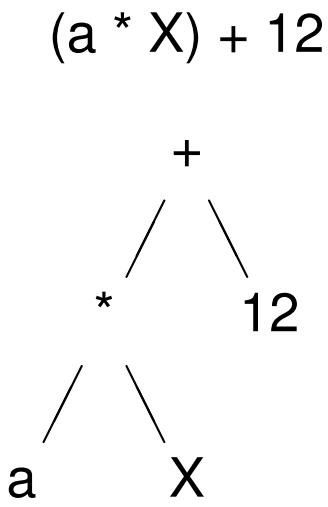
Strukturen (zusammengesetzte Terme)

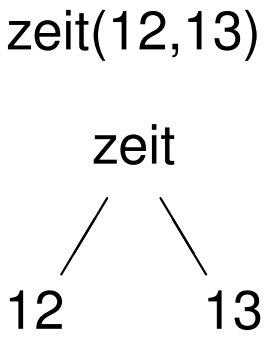
- haben eine Funktor (ein Atom) und eine bestimmte Anzahl an Argumenten (Stelligkeit)
  $\mathtt{f}\,(t_1, \ldots, t_n)$: f/n, n-stelliger Funktor, t_i Term

Weitere Begriffe

- Atome und Zahlen werden auch ***atomare Terme*** genannt
  Prolog-Prädikat zum Test auf atomare Terme: atomic/1
- ein Term ohne freie Variablen heißt ***Grundterm***
  Prolog-Prädikat zum Test auf Grundterme: ground/1

Terme als Bäume





Syntax von Prolog

Fakten

- Fakten sind Terme, auf die ein "." folgt.
  Fakten werden benutzt um auszudücken, daß etwas unbedingt wahr ist.

```
groesser(elefant, pferd).
```

Regeln

- Regeln bestehen aus einem *Kopf* und einem *Rumpf*, die durch `:-` getrennt sind. Der Kopf der Regel ist wahr, falls alle Prädikate im Rumpf als wahr bewiesen werden.
- Das '`,`' drückt eine Konjunktion aus.

```
grossvater(X, Y) :-
            vater(X, Z),
            elternteil(Z, Y).
```

Syntax von Prolog

Programm

- Fakten und Regeln werden auch Klauseln genannt. Ein Prologprogramm besteht aus einer Menge von Klauseln.

Prozedur

- Eine Prozedur besteht aus all den Klauseln, die sich im Kopf auf dieselbe Relation beziehen.

Anfragen

- Anfragen sind Terme (oder Folgen von Termen) gefolgt von einem "." Sie werden am Systemprompt eingegeben und führen zu einer Antwort des Prologsystems.

```
?- ist_groesser_als(pferd, X), ist_groesser_als(X,
hund).

X = esel;
Yes.
```

Einige Built-In-Prädikate für Terme

=.. `?Term =.. ?Liste`

Zerlegt oder erzeugt einen `Term` in/aus einer `Liste`

f($t_1, \ldots t_n$) =.. [f, $t_1, \ldots t_n$]

```
?- f(g(a),17) =.. L.
L = [f, g(a), 17]
?- X =.. [g, 12, 38].
X = g(12, 38)
```

functor `functor(?Term, ?Funktor, ?Arität)`

`Term` hat `Funktor` mit `Arität`

```
?- functor(f(a,g(a),b), F, A).
F = f
A = 3
?- functor(T, g, 4).
T = g(_G291, _G292, _G293, _G294)
```

Einige Built-In-Prädikate für Terme

arg `(?I, ?Term, ?Teilterm)`

`I`ter `Teilterm` von `Term`

arg(i, f($t_1, \ldots t_n$), t_i)

```
?- arg(2,f(a,g(a),b), T).

T = g(a)

?- arg(2, T, a).

ERROR: arg/3: Arguments are not sufficiently instantiat

?- arg(I, f(a,a,b), a).

I = 1 ;
I = 2 ;
No
```

Unifikation

Unifikation

- Unifikation bedeutet, Terme zu vereinheitlichen
- untersucht ob zwei Terme unifizieren (“zueinander passen”)
- Zwei Terme unifizieren, wenn sie identisch sind oder durch ***Variableninstantiierungen*** identisch gemacht werden können.
- gesucht wird ein *allgemeinster* Unifikator

Wichtig

- Eine gleiche Variable muß immer mit dem gleichen Wert in einem Ausdruck instantiiert werden

Unifikationsverfahren

- Unifiziere zwei Terme S und T wie folgt:
  1. Sind S und T Atome oder Zahlen, so unifizieren sie genau dann, wenn sie identisch sind
  2. Wenn S eine Variable ist, so unifizieren S und T und S wird an T gebunden
  3. Wenn T eine Variable ist, so unifizieren T und S und T wird an S gebunden
  4. Wenn S und T Strukturen sind, unifizieren sie, falls
     (a) S und T den selben Funktor mit derselben Arität besitzen und
     (b) die Argumente paarweise unifizieren

Unifikationsregeln (Überblick)

Term1	Term2	Aktion
Variable	Variable	Term1 $<=> Term2$
Variable	keine Variable	Term1 $<= Term2$
keine Variable	Variable	Term2 $<= Term1$
keine Variable	keine Variable	Unifiziere Funktor und gehe rekursiv in Struktur

Unifkation durch Prolog

- Das Prologsystem kann explizit gefragt werden, ob zwei Terme unifizieren
  Syntax in Prolog: **=**

```
?- geboren(anton,ulm) = geboren(anton,X).
X = ulm
Yes.
```

Beispiele:

f(X,a)=f(a,X).
tom=tom.
2 + 1 = 3.
mag(jane,X) = mag(X,jim).
f(a, g(X, Y)) = f(X, Z), Z = g(W,h(X)).
p(X, 2, 2) = p(1, Y, X).

Prolog-Prädikat zum Unifizieren: =

Achtung, im Gegensatz dazu == und \==: Syntaktische (Un)Gleichheit von Termen

Ein paradoxes Verhalten von Prolog

Was passiert bei Unifikation von $X = f(X)$?

$\longrightarrow$ klarerweise nicht unifizierbar

Prolog jedoch:

```
?- X=f(X).

X = f(f(f(f(f(f(f(f(f(f(...)))))))))).
```

Haben wir den ersten Implementierungsbug gefunden???

Occurs-Check

- Unifikation ist sehr teuer
- Analyse: Hauptverursacher ist der Check, ob eine Variable in dem anderen Term vorkommt: ***Occurs-Check***
- Trick: Schalte diesen Test aus

$\longrightarrow$ Prolog wird jetzt schön schnell

$\longrightarrow$ aber leider auch falsch (zumindest nach unserer Definition der Unifikation)

- In unserem Beispiel wird nun falscherweise x mit f(X) unifiziert
- Das ist nicht weiter schlimm, es entsteht im Speicher eine rekursive Struktur
  - Das kann man übrigens zur effizienten Programmierung von Verkettungen benutzen
- Probleme gibt's erst beim Versuch, diese Struktur auszudrucken

- ordentliche Unifikation: unify_with_occurs_check/2

Abarbeitung von Anfragen

Anfragen

- Eine Anfrage bedeutet, daß das ***Ziel***, das durch die Anfrage repräsentiert wird, bewiesen (***erfüllt***) werden muß.

  Dies durch das Programm, das momentan in der Wissensbasis ist.

Prinzip

- Abarbeitung einer Anfrage geht von der Anfrage selbst aus
- Anfrage wird mit Hilfe von Klauseln auf einfachere Aussagen vereinfacht, bis diese Fakten des Prolog-Programms sind.

Abarbeitungsregeln

- Wenn ein Ziel mit einem Fakt unifiziert, ist es erfüllt.
- Wenn ein Ziel den Kopf einer Regel unifiziert, dann ist es erfüllt, wenn der Rumpf der Regel erfüllt ist.
- Wenn ein Ziel aus mehreren durch Kommata getrennte Teilzielen besteht, ist es erfüllt, falls alle Teilziele erfüllt sind.

Anmerkung: Es gibt bereits eingebaute Prädikate, hierfür bestimmt das System, wann diese erfüllt sind

Abarbeitungsreihenfolge

Abarbeitung der Teilziele

- Teilziele werden *von links nach rechts* abgearbeitet, d.h. zu erfüllen versucht.
- ist ein Teilziel erfüllt, wird nächstes abgearbeitet.

Reihenfolge des Matchen der Klauseln

- Für ein Teilziel wird eine passende Klausel gesucht.
- *Reihenfolge der Durchsuchung entspricht der der Klauseln in der Wissensbasis, d.h. im Programmfile*
- Reihenfolge der Klauseln hat also Einfluss, welche Lösungen zuerst gefunden werden.

Backtracking

- Suchprozess kann in eine Sackgasse führen, d.h. eine ausgewählte, passende Klausel führt nicht zum Ziel.
  $\rightsquigarrow$ eine spätere, alternative, ebenfalls passende Klausel muß versucht werden.
- Damit verbundenes Zurücksetzen des Suchprozeß nennt man ***Backtracking***.

Beispiel

- Alle Menschen sind sterblich.
- Socrates ist ein Mensch.
  $\rightsquigarrow$ Deshalb ist Socrates sterblich

Übersetzung in Prolog

```
mortal(X) :- man(X).
man(socrates).
```

```
?- mortal(socrates).
Yes.
```

Beispiel

```
mag(mary,speisen).
mag(mary,wein).
mag(john,wein).
mag(john,mary).
```

```
?- mag(mary,X), mag(john,X).
   Call: (8) mag(mary, X)
   Exit: (8) mag(mary, speisen)
   Call: (8) mag(john, speisen)
   Fail: (8) mag(john, speisen)
   Redo: (8) mag(mary, X)
   Exit: (8) mag(mary, wein)
   Call: (8) mag(john, wein)
   Exit: (8) mag(john, wein)

X = wein

Yes
```

Anmerkung: der Verfolgungsmodus für obige Ausgabe wurde angestellt mittels

```
?- trace.
```