Logische Programmierung in Prolog II

- **Termgleichheit**
- **Beweisstrategie**
- **Listen in Prolog**
- **Arithmetik**

Unifikation

- **Problem**
  - Wird eine Frage gestellt, so muss das Prolog-System entscheiden, ob der Term der Frage zu einem Term der Wissensbasis gleich ist oder nicht.
  - Nach Einführung von Termen mit Funktionssymbolen ist ein Mechanismus zum Auffinden „passender“ Regelköpfe zu definieren.
- **Prolog benutzt die *Unifikation*, um zu prüfen, ob zwei Terme miteinander übereinstimmen**
  - Zwei Terme sind ***unifizierbar***, falls es eine Substitution gibt, die die Terme gleich macht
  - Diese Substitution wird ***Unifikator*** genannt.
- **Beispiele**
  - t1 = `datum(D,M,1983)`, t2 = `datum(D1,may,Y1)`

    Die Substitution S = {D = D1,M = may, Y1= 1983} unifiziert t1 und t2

Termgleichheit

Regeln für Termgleichheit

- **Zahlen und Atome**
  - sind nur zu sich selbst gleich
- **Variable**
  - ist gleich zu einer anderen Variablen, wenn die Namen gleich sind.
  - Variablen unifizieren mit jedem Term.
  - Wird eine Variable durch einen Term substituiert, dann ist diese mit dem Term instanziiert.
  - Alle Vorkommen der Variablen werden durch den Term ersetzt.
- **Strukturen**
  - Zwei Strukturen sind gleich, wenn
    - sie den gleichen Funktor haben
    - die gleiche Anzahl von Komponenten haben
    - die entsprechenden Komponenten der ersten und zweiten Struktur paarweise gleich sind.

Termgleichheit

Beispiele Unifikation

- **Beispiele**
  - T1 = f (X, a(b, c))
    T2 = f(d, a (Z, c))
  - T1 = f(X, a(b, c))
    T2 = f(Z, a(Z, c))
  - T1 = f(c, a(b, c))
    T2 = f(Z, a(Z, c))
  - T1 = g(Z, f(A, 17, B), A+B, 17)
    T2 = g(C, f(D, D, E), C, E)

Substitution

- **Substitution**
  - Eine Substitution ist eine endliche Menge von Paaren der Form X=t
  - X ist eine Variable, t ein Term
  - Es gilt: Keine zwei Paare haben dieselbe Variable als linke Seite
- **Instanz eines Terms**
  - Für einen Term t und seine Substitution S ={X1 = t1 , ... Xn = tn} bezeichnet tS das Ergebnis der simultanen Ersetzung aller Vorkommen aller Xi durch ti (1<=i<=n).
  - tS heißt Instanz von t.
- **Anmerkungen**
  - Ein Ziel und ein Fakt (ein Regelkopf) „passen“, wenn sie eine gemeinsame Instanz besitzen.
  - Eine Instanz heißt Grund-Instanz, wenn sie keine Variablen enthält.
  - Die Berechnung eines PROLOG-Programms liefert, ausgehend von der Frage, die Substitutionen, für die alle Ziele der Frage erfüllt sind.

Unifikator

- **Unifikator**
  - Ein Unifikator zweier Terme ist eine Substitution, die die beiden Terme identisch macht, also eine Instanz erzeugt, die Instanz beider Terme ist. In diesem Fall sagt man, dass die beiden Terme unifizierbar sind.
- **Beispiel:**

p(X,	f(g(c,d),	Y),	Z,	W)
p(g(c,d),	f(X,	b),	b,	V)

haben als Unifikator die Substitution

S = { X/g(c,d), Y/b, Z/b, W/V }

das heißt, es gilt:

S(T1) = S(T2)

Termgleichheit

Allgemeinster Unifikator

- **Eine Instanziierung, mit der die Terme gleich werden, kann allgemeiner sein als eine andere**
  - z.B. für `date(D,M,1983)` und `date(D1,may,Y1)` ist die Instanziierung
    - S1 = {D /D1, M/ may, Y1/1983}
  - allgemeiner als die Instanziierung
    - S2 = {D/3 , M/may, Y1/1983}
  - da nicht alle Komponenten festgelegt sind.
  - Term t_1 ist allgemeiner als Term t_2, wenn t_2 eine Instanz von t_1 ist.
- **Der Allgemeinste Unifikator**
  - für zwei Terme ist ein Unifikator, der die allgemeinste Instanz erzeugt.
- **Satz**
  - Wenn zwei Terme unifizierbar sind, dann existiert ein eindeutiger allgemeinster Unifikator, ggfs. in alphabetischen Varianten (Umbenennung von Variablen).

Termgleichheit

Algorithmus MGU

```
Eingabe: Terme t1 und t2
Ausgabe: MGU oder "nicht unifizierbar"

IF t1 und t2 gleiche Variablen THEN  U = {}

IF t1 eine Variable THEN
  IF t2 enthält nicht t1 THEN  U = { t1/t2 }
  ELSE nicht unifizierbar

IF t2 eine Variable THEN
  IF t1 enthält nicht t2 THEN  U = { t2/t1 }
  ELSE nicht unifizierbar

IF t1 und t2 sind Konstanten THEN
  IF t1 = t2 THEN              U = {}
  ELSE nicht unifizierbar

IF t1 und t2 Strukturen THEN
  IF Funktoren und Anzahl Komponenten sind gleich THEN
    Sei U1 = MGU (t11, t21)
    FOR alle Komponentenpaare t1i, t2i 2<=I<=n DO
      Ui := MGU( Ui-1(..(U1(t1i))..), Ui-1(..(U1(t2i))..) )
    IF alle Ui existieren THEN      U = Un(Un-1(..(U1))..)
    ELSE nicht unifizierbar
ELSE nicht unifizierbar
```

Beispiele MGU

- **Beispiel**
  - T1 = g(f(1), h(X))
    T2 = g(Y, Y)
  - T1 = p(1, A, f(g(X)))
    T2 = p(X, f(Y), f(Y))
  - T1 = p(X, f(g(c, d), Y), Z, W)
    T2 = p(g(c, d), f(X, b), V, h(X,Z))

Variablen

- **Variablen in Prolog werden anders gebraucht als in herkömmlichen Programmiersprachen**
  - Variablen referenzieren Objekte, nicht auf Speicherplätze
  - sie werden bei der Unifikation instanziiert
- **Der Operator = bedeutet nicht Zuweisung, sondern Unifikation**
  Bsp:
  - X = 3. bedeutet, dass die Variable X unifiziert wird mit der Konstante 3 und dabei mit 3 instanziiert wird.
  - = kann nicht verwendet werden, um den Wert einer Variable zu erhöhen
    d.h. X = X + 1. erzeugt einen Fehler

Termgleichheit

Gleichheit von Termen

- **Gleichheit**
  - von Termen bedeutet, dass diese unifizierbar sind
- **Systemprädikat =**
  - = (X,X)
  - dieses Faktum ist jedem Prolog-System bekannt
  - Es ist nicht erlaubt, neue Klauseln zu diesem Prädikat hinzuzufügen
  - Schreibweise:
    - = (t1, t2)
    - t1 = t2
  - Bedeutung;
    - t1= t2 ist dann beweisbar, wenn die Terme unifizierbar sind
- **Beispiel**
  - 3+2=5 ist nicht beweisbar
  - weiblich(susanne) = weiblich(X) ist beweisbar

Beweisstrategie

Resolutionsprinzip - 1

- **Prolog versucht eine Anfrage auf der Menge der vorhandenen Fakten und Klauseln zu beweisen.**
  - Ein Faktum ist eine beweisbare Aussage
  - Es gilt
    - wenn jedes der Literale L1 , L2, L3, …, Ln beweisbar ist
    - und es eine Regel L :- L1 , L2, L3, …, Ln gibt
    - dann ist auch das Literal L beweisbar.
  - Beispiel:

```
istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

  - Modus Ponens (Abtrennregel)
- **Resolutionsprinzip**
  - ist die Umkehrung des Modus Ponens
  - liefert ein Verfahren, um zu zeigen, ob eine Aussage beweisbar ist oder nicht.

Beweisstrategie

Resolutionsprinzip - 2

- **Eine Aussage ist beweisbar,**
  - wenn sie ein Faktum ist
  - wenn mindestens eine Regel existiert, deren linke Seite identisch mit der zu beweisenden Aussage ist
  - und deren Literale der rechten Seite alle beweisbar sind.
- **Beispiel**
  - `istVaterVon(klaus, aline)` V/klaus, K/aline
    wird zurückgeführt auf
  - `verheiratet(klaus,F), istMutterVon(F,aline)` F/susanne
    wird zurückgeführt auf
  - `verheiratet(klaus,susanne)` ist ein Faktum, bewiesen
  - `istMutterVon(F,aline)` ist ein Faktum, bewiesen
    wird zurückgeführt auf
  - leere Aussage

Beweisstrategie

Resolutionsprinzip - 3

- **Satz**
  - Kann eine Aussage in mehreren Schritten unter Verwendung des Resolutionsprinzips auf die leere Aussage zurückgeführt werden, dann ist diese Aussage beweisbar.
- **Folgende Ergebnisse sind möglich**
  - yes
    - Die gegebene Aussage ist aufgrund der Fakten und Regeln beweisbar.
  - No
    - Die gegebene Aussage ist aufgrund der Fakten und Regeln nicht beweisbar. Möglicherweise ist die Aussage korrekt, nur sind die Regeln und Fakten unvollständig.
  - hält nicht an
    - Die Aussage könnte aufgrund der Regeln und Fakten beweisbar sein.

Beweisstrategie

- **Resolution ist keine eindeutige Vorschrift**
  - Welches Literal einer Anfrage soll als nächstes bewiesen werden?
  - Durch welche Klausel soll ein Literal einer Anfrage abgeleitet werden?
- **Strategie**
  - A) Die Literale einer Anfrage werden von links nach rechts abgeleitet.
  - B) Die Klauseln werden von "oben" nach "unten" nach einer passenden durchsucht.
  - C) Falls das Prolog-System beim Beweisen in eine Sackgasse läuft, wird der letzte Ableitungsschritt rückgängig gemacht und die nächste passende Klausel zu einem neuen Ableitungsschritt verwendet (backtracking).

    Bem backtracking müssen Bindungen von Variablen, die beim Beweisschritt stattgefunden haben, rückgängig gemacht werden, damit neue Variablenbindung im alternativen Beweisschritt versucht werden kann.

Backtracking -Beispiel - 1

```
verheiratet(werner, monika).
verheiratet(gerd, renate).
verheiratet(klaus, susanne).

istMuttervon(susanne, aline).
istMutterVon(susanne, dominique)
istMutterVon(monika, karin).
istMutterVon(monika, klaus).
istMuttervon(renate, peter).
istMuttervon(renate, susanne).

istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

Beweisstrategie

Backtracking -Beispiel - 2

istVaterVon(X,aline)

```
istVaterVon(X,aline) :- verheiratet(X,F),    K/aline, V/X
                        istMutterVon(F,aline).


verheiratet(X,F)                          X/werner , F/monika
istMutterVon(monika, aline)               FAIL


verheiratet(X,F)                          X/gerd , F/renate
istMutterVon(renate, aline)               FAIL


verheiratet(X,F)                          X/klaus , F/susanne
istMutterVon(susanne, aline)

X=klaus
```


Listen

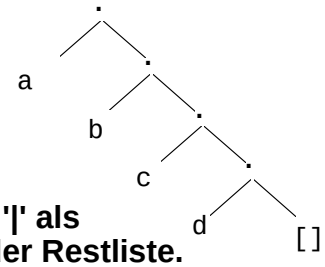
Listen in Prolog

- **Eine Liste ist eine Folge von Objekten.**
  - Die Anzahl der Objekte einer Liste ist nicht festgelegt.
- **Eine Liste in Prolog ist entweder**
  - die leere Liste (dargestellt durch das Atom [])
  - die Struktur mit dem Funktor **.** und zwei Komponenten
    - die erste Komponente wird head genannt
    - die zweite Komponente ist wiederum eine Liste (tail)
- **Beispiele für Listen:**
  - []
  - . (10, [])
  - . (x, . (y, []))
  - . (a, . (b, . (c, [])))

Listen

Listen repräsentiert als Bäume

- **Listen können als Bäume repräsentiert werden.**
  - . (a, . (b, . (c, .(d, [])))) Listennotation
  - Die Elemente werden durch Komma getrennt und mit eckigen Klammern umschlossen.
  - [a, b, c, d]



- **Prolog verwendet den vertikalen Strich '|' als Trennsymbol zw. dem Listenkopf und der Restliste.**
- **alternative** Repräsentationen können verwendet werden:
  - [elem1,elem2,elem3] oder
    [head | tail] oder
    [elem1 , elem2, … | rest] oder …
  - z.B. [a,b,c] = [a | b,c] = [a , b | c] = [a,b,c, []]

Listen

Listen in Prolog

- **Unifikation**
  - Listen sind spezielle Terme
  - Länge der Liste entspricht der Stelligkeit
  - Listen können miteinander unifiziert werden.
- von Listen ergibt:

L1	L2
[a,b,c]	[X,Y,Z]
[a,b,c]	[X|Y]
[a]	[X|Y]
[[a, Y] | Z]	[[X, b], [c, d]]
[X, Y, X]	[a, Z, Z]
[[X], [Y], [X]]	[[a], [Z], [Z]]

Operationen auf Listen: member

- **Prolog-Prädikate (Prozeduren) werden oft durch**
  - einen Basisfall und
  - einen einfachen rekursiven Fall definiert:
- **Beispiel:** die Relation member
  - Ein Element ist in einer Liste, falls
    - es das erste Listenelement ist **(Basisfall)** oder
    - es in der Restliste ist **(rekursiver Fall)**
-
```
member(X,[X|_]).

member(X,[_|Y]) :- member(X,Y).
```


Operationen auf Listen: member

```
?- member(1,[1,2,3]).
yes

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no

?- member(1,[a,b]).
   (3) 0 Call: member(1,[a,b]) ?
   (4) 1 Call: member(1,[b]) ?
   (5) 2 Call: member(1,[]) ?
   (5) 2 Fail: member(1,[]) ?
   (4) 1 Fail: member(1,[b]) ?
   (3) 0 Fail: member(1,[a,b]) ?
no
```

```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).
```

Listen

Operationen auf Listen: member

```
?- member(X,[a,c]),member(X,[b,c]).
   (5) 0 Call: member(\_35,[a,c]) ?
   (5) 0 Exit: member(a,[a,c]) ?
   (6) 0 Call: member(a,[b,c]) ?
   (7) 1 Call: member(a,[c]) ?
   (8) 2 Call: member(a,[]) ?
   (8) 2 Fail: member(a,[]) ?
   (7) 1 Fail: member(a,[c]) ?
   (6) 0 Fail: member(a,[b,c]) ?
   (5) 0 Redo: member(a,[a,c]) ?
   (9) 1 Call: member(\_35,[c]) ?
   (9) 1 Exit: member(c,[c]) ?
   (5) 0 Exit: member(c,[a,c]) ?
   (10) 0 Call: member(c,[b,c]) ?
   (11) 1 Call: member(c,[c]) ?
   (11) 1 Exit: member(c,[c]) ?
   (10) 0 Exit: member(c,[b,c]) ?
X = c
```

```
member(X,[X|_]).

member(X,[_|Y]):-
   member(X,Y).
```


Listen

Operationen auf Listen: append

Beispiele:

append([a,b],[c,d],[a,b,c,d]). append([a,b,d],[c],[a,b,d,c]).

append([],[c,d],[c,d]).

- **Idee:** Unterscheidung nach dem ersten Argument von append
  - **Basisfall:** falls das erste Argument von append die **leere Liste** ist, dann enthält das dritte Argument die gleiche Liste wie das zweite Argument
  - **rekursiver Fall:** falls das erste Argument von append **nicht** die leere Liste ist, dann muß es die Form **[X|L1]** haben.

Daher hat das dritte Argument die Form [X|L3] , wobei L3 die aneinander gehängten Listen L1 und L2 enthält.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Listen

Operationen auf Listen: append

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

?- append([a,b],[c,d],L3).
L3 = [a,b,c,d]

?- append(L1,L2,[a,b]).
L1 = [],
L2 = [a,b] ;
L1 = [a],
L2 = [b] ;
L1 = [a,b],
L2 = [] ;
no
```


Arithmetik

Operatoren & Arithmetik

- **Prolog erlaubt die Verwendung von Operatoren.**
- **Operatoren werden in Infix-Notation geschrieben.**
  - z.B. 1+3 statt +(1,3)
- **Grundlegende arithmetische Operatoren sind**

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
mod	Modulo

- **Vergleichsoperatoren**

X > Y	X ist größer als Y
X >= Y	X ist größer als oder gleich Y
X < Y	X ist kleiner als Y
X =< Y	X ist kleiner als oder gleich Y
X =:= Y	X ist gleich Y
X =\= Y	X ist ungleich Y

Arithmetik

Arithmetik: 'is' und '='

- **Der Operator =**
  - Er bedeutet, daß Unifikation verwendet wird.
  - Die Unifikation gelingt, wenn der Term auf der rechten Seite von = mit dem Term der linken Seite unifiziert werden kann.
    ?- X = 3+1.
    X = 3+1.
    ?- 4 = 3+1.
    no
    ?- 3+1 = 3+1.
    yes
- **Der Operator is**
  - ruft die arithmetische Evaluierung der rechten Seite auf und versucht, das Ergebnis mit der linken Seite zu unifizieren.
    ?- X is 3+1.
    X = 4
    ?- 4 is 3+1.
    yes
    ?- 3+1 is 3+1.
    no

Arithmetik

Arithmetik

- **Beispiel: Länge einer Liste length(L1,N),d.h.**
  - falls L1 leer ist,dann hat N den Wert 0,
  - sonst ist N gleich der Länge des Tails von L1 + 1
- **Prädikat length**
  - `length([],0).`
  - `length([H|T],N1) :- length1(T,N),`
    `N1 is N + 1.`
- **Beispiele**
  - `?- length([a, b, c, d], X).`
    `X = 4`
  - `?- length([a,b], X), Y is 4 + X.`
    `X = 2`
    `Y = 6`

Deklarative vs. Prozedurale Interpretation

- **Deklarative Bedeutung**
  - Sie bezieht sich auf die logischen Relationen, die durch das Programm beschrieben werden, d.h. WAS ist das Ergebnis des Programms
- **Prozedurale Bedeutung**
  - Sie bezieht sich darauf, WIE das Ergebnis berechnet wird.
- **Deklarative Programmierung ist einer der Vorteile von Prolog, denn die Idee ist einfach:**
  - Gebe Relationen an, die für eine Applikation gelten müssen, und kümmere dich nicht darum, WIE die Lösung gefunden wird.
- **Aber: der prozedurale Aspekt ist wichtig,**
  - z.B. die Regel p:-p. ist deklarativ korrekt, aber prozedural sinnlos, da sie eine endlose Schleife bewirkt.