Zusammenfassung Prolog Repititorium

Def: Ein Prolog Programm ist eine geordnete Menge von Klauseln.
Klauseln können entweder Fakten oder Regeln sein.

Fakt: mann(tobias). vater(frank, tobias).
Regel: grossvater(X,Y) :- vater(X,Z), vater(Z,Y). (X,Y,Z sind Variablen)

Fakten sind von Natur aus wahr.
Der Kopf einer Regel gilt, wenn alle Bedingungen des Regelrumpfes erfüllt sind.

Um eine Regel zu beweisen, beweise alle Ziele dieser Regel.
grossvater(X,Y) :- vater(X,Z), vater(Z,Y)
konklusion ← prämisse ∧ prämisse

Um eine Anfrage zu beantworten sucht der Prolog Interpreter im gegebenen Programm von oben nach unten nach einer Klausel, deren Kopf sich mit der Anfrage syntaktisch zur Deckung bringen lässt.
Handelt es sich um ein Faktum ist der Beweis gelungen.
Bei einer Regel wird in der Anfrage der Regelkopf durch den zugehörigen Regelrumpf ersetzt, dessen Teile dann nacheinander von links nach rechts auf die gleiche Weise zu beantworten versucht.

Die Bindung von Variablen in Prolog heisst **Unifikation.** Diese Operation versucht zwei Terme identisch zu machen, indem Variablen an ihre Gegenüber gebunden werden. (Beachte: Variablen auf beiden Seiten werden gebunden)

Prüfe: Name d. Prädikats
Stelligkeit
Unifiziere

Unifikation: das Angleichen zweier Terme durch gültige
Variablensubstitutionen

Seien t_1 und t_2 Terme.
t_1 und t_2 sind unifizierbar, falls es eine Substitution s gibt mit $s(t_1) = s(t_2)$

Der Unifizierungsalgorithmus sucht die allgemeinste Lösung d.h. Die Substitution die am wenigsten einschränkt. Diese Lösung wird als allgemeinster Unifikatior bezeichnet.

Bsp:

2+1 = 3 FAIL (die arithmetische Operation wird vor der Unifizierung nicht ausgewertet.

f(X,a) = f(a,X) SUCCEEDS X=a

likes(jane,X) = likes(X,jim) FAIL X=jane, daher kann X nicht mehr mit jim belegt werden.

f(X,Y) = f(P,P) SUCCEEDS X=Y=P

s(X,X,Y) = s(2,3,4) FAIL X kann nicht an 2 und 3 gebunden werden

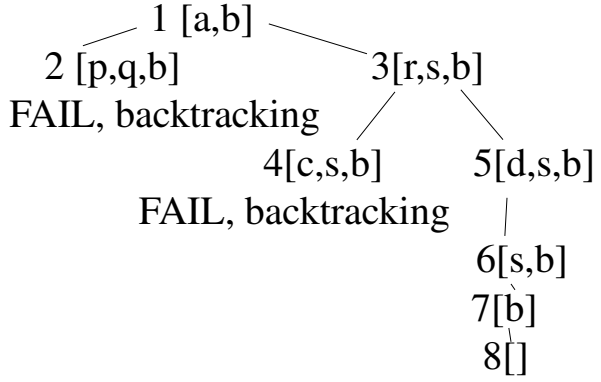
s(X,X,Y) = s(2,2,4,5) FAIL unterschiedliche Stelligkeit

s(X,X,Y) = s(2,T,4) SUCCEEDS X=T=2

Beweissuche:
R1: a:-p,q
R2: a:-r,s
R3: b.
R4: d.
R5: r:-c.
R6: r:-d.
R7: s.

Anfrage: a,b

Suchbaum:



Erweiterte Methode: Rekursion
Rekursion wird in Prolog verwendet um Schleifen zu konstruieren.
Man braucht immer zwei Regeln, eine die die Rekursion beendet, und eine die die Rekursion selber enthält.

Bsp: Vorfahren finden:

vorfahr(X,Y) :- elternteil(X,Y). ← einfachster Fall = Abbruchfall

vorfahr(X,Y) :- elternteil(X,Z), vorfahr(Z,Y).

Beachte: Um Endlosschleifen zu vermeiden steht der Rekursionsfall im Regelrumpf meist ganz rechts. → Endrekursion

Wege in kreisfreien Graphen:

```
kante(1,2).          weg(X,Y) :- kante(X,Y)   Abbruchfall
kante(3,4).          weg(X,Y) :- kante(X,Z), weg(Z,Y).
kante(5,7).

bigger(cat,mouse).       bigger_than(X,Y) :- bigger(X,Y). Abbruchfall
bigger(dog,cat).         bigger_than(X,Y) :- bigger(X,Z), bigger_than(Z,Y).
bigger(sheep,dog).
```

Logische Repräsentation:

mutter(X,Y) :- kind(Y,X), weiblich(X).

$\forall X (\forall Y((kind(Y,X) \wedge weiblich(X) \rightarrow mutter(X,Y).$

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

$\forall X (\forall Y(\exists Z (vater(X,Z) \wedge vater(Z,Y) \rightarrow grossvater(X,Y).$

Variablen die im Kopf einer Regel vorkommen sind Allquantifiziert.
Variablen die ausschliesslich im Rumpf einer Regel vorkommen sind existenzquantifiziert.

Listen:

Listen dienen zur Darstellung „komplexer“ Datenstrukturen in Prolog.
Schreibweise: [1,2,3,4,5].
Um auf einzelne Elemente zuzugreifen, benutzt man den
Restlistenoperator [X|Y]. X=1, Y=[2,3,4,5]
[X,Y|Z] X=1, Y=2, Z= [3,4,5]

s([X | L],[Y | R]) = s([1,2], [3,4]) SUCCEEDS X=1, L=[2], Y=3, R=[4]
s([X | L],[Y | R]) = s([1,2], 1) FAIL
[X | X] = [1, [1]] FAIL

Operationen auf Listen:

Append:

```
append([], L, L).
append([ K | R ], L, [ K | L1 ]) :- append( R, L, L1).
```

Invert:

```
invert( [], []).
invert([ X | R ], Erg) :- invert(R, Rinv), append(Rinv, [X], Erg).
```

Man braucht für invert append, da man zum Umdrehen einer Liste die Elemente der Ausgangsliste in umgedrehter Reihenfolge auf das Ergebnis legen muss, dies ist mit den bisher gelernten Methoden (Rekursion, Restlistenoperator) nicht möglich.

Flachmachen einer Liste:

```
flatten( [], []).
flatten( X, [X]) :- atomic(X).
flatten( [X | Y], Z) :- flatten(X,X1), flatten(Y,Y1), append(X1, Y1, Z).
```

Da jedes Element der Ausgangsliste selber eine Liste sein kann müssen wir flatten für das Element und für den Rest der Liste wieder aufrufen, und die Teillisten am Ende zum Ergebnis verbinden.

Länge einer Liste berechnen:

```
listlen([],0).
listlen([ _ | R]) :- listlen( R, N1), N is N1+1.
```

Member:

```
member(X, [ X | R ]).   einfachster Fall: gesuchtes Element ist Kopf
member(X, [ _ | R ]) :- member(X,R). sonst: spalte Kopf ab, durchsuche
                                                Rest
```

Element einer Liste löschen

singuläre Liste, bzw nur erstes Vorkommen löschen

```
delete( Element, [ Element | Rest ], Rest).
delete( Element, [ Kopf | Rest ], [ Kopf | Z ]) :- delete(Element, Rest, Z).
```

```
lösche alle Vorkommen
delall(_, [], []).
delall(Element, [ Element | Rest], Erg) :- delall( Element, Rest, Erg),!.
delall(Element, [ Kopf | Rest], [ Kopf | Z ]) :- delall(Element, Rest,Z).
```

Permutation einer Liste:

```
delete( Element, [ Element | Rest ], Rest).
delete( Element, [ Kopf | Rest ], [ Kopf | Z ]) :- delete(Element, Rest, Z).

permutation([], []).
permutation(L, [ X | R ]) :- del(X, L, L1), permutation(L1,R).
```

Da Prolog endrekursive Funktionen besonders effizient berechnen kann, sind *invert* und *listlen* nicht optimal. Um solche Probleme besser lösen zu können benutzt man die Akkumulatortechnik.
Hier wird eine Hilfsvariable, der Akkumulator benutzt um Ergebnisse zu sammeln und am Schluss auszugeben.

Um die Stelligkeit des Prädikats beizubehalten definiert man Aufrufprädikate, die den Akkumulator hinzufügen:

```
invert(L,Erg) :- invert(L,[],Erg).
invert( [] ,L , L).
invert( [ X | R ], L, Erg) :- invert( R, [ X | L ], Erg).

listlen(L,N) :- listlen(L,0,N).
listlen([],A,A).
listlen( [ K | R ], A, N) :- A1 is A+1, listlen( R, A1, N ).
```

!, der cut operator:

Der cut operator wird dazu benutzt, um das Backtracking zu steuern. Ist der cut Operator (ist immer wahr) einmal passiert, kann links von ihm kein Backtracking mehr stattfinden.

Bsp: Implementation von not:

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

Da not fehlschlagen soll, sobald es eine Belegung für X gibt, ist backtracking nicht erwünscht.

Beachte: not ist kein logisches nicht, da es kein falsch gibt, sondern nur ein „nicht beweisbar“.

Trick um Belegbarkeit einer Variable zu testen ohne sie zu instatiieren:

```
not(not(s(X)).
```

Implementierung der Kontrollstruktur if then else mit Hilfe des Cuts:

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

Differenzlisten:
Nachteil von append: Die Rechenzeit steigt linear mit der Länge der ersten Liste, da diese komplett abgebaut wird, um Elementweise vorne an die zweite angehängt zu werden.
Man benutzt „unvollständige“ Listen um das ganze in konstanter Zeit (genauer durch eine einzige Unifikation) durchzuführen.

[1,2 | X] - X Man fügt der Liste etwas hinzu und zieht es wieder ab. Die Liste wird hierdurch nicht verändert.

X-Y

Y-Z

X - Z

append_dl (X – Y, Y - Z, X – Z).

Beispielaufruf:

append_dl([1,2,3 | A] - A, [4,5 | B] - B, Erg - []).

X = [1,2,3 | A] → X = [1,2,3 | 4,5 | B]

Y = A = [4,5 | B]

Z = B = []

X = [1,2,3,4,5]

DCG:
Da Differenzlisten im Gegensatz zu append effektiver im Verketten und Wesentlich Effektiver im zerlegen von Listen sind, benutzt man diese Technik um Grammatiken und Parser zu schreiben.

Der Differenzlistenoperator → übernimmt hierbei das Hinzufügen der Differenzlisten.

s → np, vp. ist äquivalent zu s (P0 – P) :- np (P0 – P1), vp (P1 – P).

Beispiel für Parsen und Generieren von Sätzen:

Differenzlisten und DCG

Exkurs 1: Das Türme von Hanoi Rätsel in Prolog: LINK

Exkurs 2: Sortieren von Listen
Anwendung von Listenoperationen: Sortieren von Listen.

Um eine Liste zu sortieren (Bubblesort) :

- Finde zwei benachbarte Elemente X u. Y in der Liste, so daß X > Y,
- dann vertausche X und Y und erhalte Liste1
- danach sortiere Liste1
- gibt es kein Paar X,Y ist die Liste sortiert.

```
gt(X,Y) :- X > Y.
vertausche( [ X, Y | Rest], [ Y,X | Rest]) :- gt(X,Y).
vertausche( [ Z | Rest ], [ Z | Rest1] ) :- vertausche ( Rest, Rest1).
bubblesort(Liste, Sortiert) :- vertausche( Liste, Liste1), !,
bubblesort(Liste1,Sortiert).
```

QuickSort:

Teile die Liste an einem willkürlichen Punkt. Alle größeren Elemente werden in eine, alle kleineren in eine andere Liste gesteckt.
Auf diese neuen Listen wende wieder den QuickSort Algorithmus an, bis es nichts mehr zu sortieren gibt. Dann werden die sortierten Listen zur Ergebnisliste verbunden.

```
quicksort([], []) :- !.
quicksort( [Head | Tail ], SortierteListe) :-
        split(Head, Tail, Kleiner, Groesser),
        quicksort(Kleiner, KleinerSortiert),
        quicksort(Groesser,GroesserSortiert),
        append(KleinerSortiert, [ Head | GroesserSortiert], SortierteListe).
```

```
split( _ , [], [], []) :- !.
split( Element, [Head | Tail], [Head | Kleiner], Groesser) :-
        Head < Element, !, split(Element, Tail, Kleiner, Groesser).
split( Element, [Head | Tail], Kleiner, [Head | Groesser]) :-
        split(Element, Tail, Kleiner, Groesser).
```

Merge Sort:
Der Algorithmus teilt die Eingabe Liste immer wieder auf, bis die Teillisten sortiert sind (spätestens wenn jedes Element alleine in einer Liste ist). Die Teillisten werden dann zur Ergebnisliste verschmolzen.

```
%mergesort( Liste, SortierteListe).
  mergesort( [], [] ) :- !.
  mergesort( [Element], [Element]) :- !.
  mergesort( Liste, Sortierte) :- divideList(Liste, Liste1, Liste2),
                                  mergeSort(Liste1, Sortiert1),
                                  mergeSort(Liste2, Sortiert2),
                                  mergge(Sortiert1, Sortiert2, Sortiert).

  mergge( [], Liste, Liste) :- !.
  mergge( Liste, [], Liste) :- !.
  mergge( [K1 | R1], [K2 | R2 ], [ K1 | R3 ]) :- K1 =< K2, !,
                                                  mergge(R1,[ K2 | R2 ] ,R3).
  mergge( L1, [ K | R1 ], [K | R2 ]) :- mergge(L1, R1,R2).

  divideList([],[],[]).
  divideList([Element],[Element],[]).
  divideList([E1, E2 | R1 ], [E | R2 ], [E2 | R3] ) :- divideList(R1, R2, R3).
```

QuickSort
teilt die Liste in 2 Teillisten, alle Elemente die grösser als ein vorher bestimmtes Pivot Element sind kommen in eine Liste, alle die kleiner sind in eine andere.

```
pivoting( H, [], [], []).
pivoting( H, [K | R], [K | S], G) :- K =< H, pivoting( H, R, S, G ).
pivoting( H, [K | R], S, [K | G]) :- K>H, pivoting(H, R, S, G).
```

```
quicksort(L, Sl) :- quicksort(L, [], Sl).
quicksort([], A,A).
quicksort([ K | R], A, Sortiert) :- pivoting(K, R, L1, L2),
                                    quicksort(L1, A, Sortiert1),
                                    quicksort(L2, [K | Sortiert1], Sortiert).
```

Insert Sort:
Der Algorithmus entnimmt der Eingabeliste ein Element, sucht in der Ausgabeliste den richtigen Ort und fügt es dort ein.

```
insert(X, [K | R], [K | Z]) :- X>K, insert( X, R, Z).
insert(X, [K | R], [X, K | R]) :- X =< K.
insert(X, [], [X]).

insertsort(L, Sl) :- insertsort(L, [], Sl).
insertsort([], A, A).
insertsort([K | R], A, Sortiert) :- insert(H,A,Na), insertsort(R,Na,Sortiert).
```