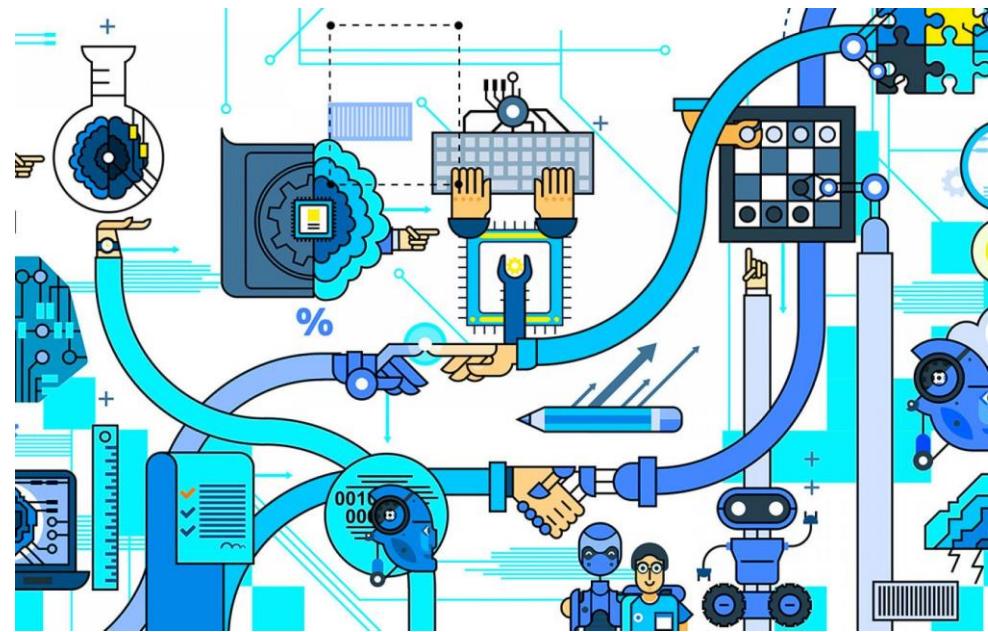


# Logische Programmierung

## Einführung in PROLOG



# Rekursion

- Ein wichtiges Konzept für das Lösen von Aufgaben bzw. für die Definition mächtiger Prädikate ist die Rekursion.
- Ein Prädikat ist rekursiv definiert, wenn in einer der definierenden Regeln das Prädikat im Regelkörper aufgerufen wird.
- Rekursion ist eine Problemlösungsstrategie. Die Grundidee ist das Zurückführen einer allgemeinen Aufgabe auf eine einfachere Aufgabe derselben Klasse (Schleifen).
- Rekursion ermöglicht es kompakte Prädikatsdefinitionen zu schreiben und Redundanz zu vermeiden.



# Rekursive Definitionen

## Beispiel: natürliche Zahlen

- 0 ist eine natürliche Zahl. (Ankerregel)
- Wenn  $n$  eine natürliche Zahl ist, dann ist auch der Nachfolger von  $n$  (also  $n + 1$ ) eine natürliche Zahl. (rekursive Regel)
- Nichts sonst ist eine natürliche Zahl. (Ausschlussregel)

## Beispiel: transitive Relation “Vorfahr”

A ist ein Vorfahr von B, wenn

- A ein Elternteil von B ist. (Ankerregel)
- wenn A ein Vorfahr von C und C ein Elternteil von B ist.  
(rekursive Regel)
- Sonst ist A kein Vorfahr von B. (Ausschlussregel)



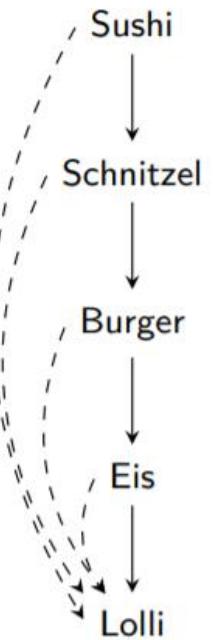
# Rekursive Prädikate in Prolog

- Ein Prädikat ist **rekursiv definiert**, wenn in einer der definierenden Regeln das Prädikat im Regelkörper aufgerufen wird.
- Das Prädikat **teurer**/2 ist rekursiv definiert.

```
kostet_etwas_mehr(eis,lolli).  
kostet_etwas_mehr(burger,eis).  
kostet_etwas_mehr(schnitzel,burger).  
kostet_etwas_mehr(sushi,schnitzel).  
  
teurer(X,Y) :-  
    kostet_etwas_mehr(X,Y).  
  
teurer(X,Y) :-  
    kostet_etwas_mehr(X,Z),  
    teurer(Z,Y).
```



# Vorteile rekursiver Prädikate



kostet etwas mehr als: —>  
ist teurer als: - - ->

```
kostet_etwas_mehr(eis,lolli).  
kostet_etwas_mehr(burger,eis).  
kostet_etwas_mehr(schnitzel,burger).  
kostet_etwas_mehr(sushi,schnitzel).  
  
% nichtrekursive Definition von teurer/2  
teurer(X,Y):-  
    kostet_etwas_mehr(X,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,B),  
    kostet_etwas_mehr(B,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,B),  
    kostet_etwas_mehr(B,C),  
    kostet_etwas_mehr(C,Y).
```



# Vorteile rekursiver Prädikate

```
kostet_etwas_mehr(eis,lolli).  
kostet_etwas_mehr(burger,eis).  
kostet_etwas_mehr(schnitzel,burger).  
kostet_etwas_mehr(sushi,schnitzel).  
  
% nichtrekursive Definition von teurer/2  
teurer(X,Y):-  
    kostet_etwas_mehr(X,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,B),  
    kostet_etwas_mehr(B,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,A),  
    kostet_etwas_mehr(A,B),  
    kostet_etwas_mehr(B,C),  
    kostet_etwas_mehr(C,Y).
```

```
kostet_etwas_mehr(eis,lolli).  
kostet_etwas_mehr(burger,eis).  
kostet_etwas_mehr(schnitzel,burger).  
kostet_etwas_mehr(sushi,schnitzel).  
  
% rekursive Definition von teurer/2  
teurer(X,Y):-  
    kostet_etwas_mehr(X,Y).  
  
teurer(X,Y):-  
    kostet_etwas_mehr(X,Z),  
    teurer(Z,Y).
```



# Deklarative und prozedurale Bedeutung einer Wissensbasis

## deklarative Bedeutung

- Unter der deklarativen Bedeutung versteht man die Bedeutung, die 'gemeint' oder die 'ausgedrückt' ist, wenn man die Wissensbasis als Menge logischer Aussagen liest.
- Die deklarative Bedeutung eines Prologprogramms kann extensional als die Menge aller Aussagen definiert werden, die sich **logisch** aus der Theorie der Wissensbasis (sprich Sammlung von logischen Aussagen) ableiten lassen.

## prozedurale Bedeutung

- Unter der prozeduralen Bedeutung versteht man die Bedeutung, die sich daraus ergibt, was Prolog mit einer Wissensbasis 'tut'.
- Die Prozedurale Bedeutung eines Prologprogramms kann extensional als die Menge aller Anfragen (Aussagen) definiert werden, für die der Prolog-Interpreter eine Variablenbelegung findet, die zu der Ausgabe **true**. führt.



# Deklarative und prozedurale Bedeutung

```
es_regnet :- es_regnet.  
es_regnet.
```

**deklarative Bedeutung:** Die Wissensbasis besteht aus zwei Aussagen: 'Wenn es regnet, dann regnet es.' und 'es regnet'. Aus diesen Aussagen lässt sich ableiten, dass es regnet.

**prozedurale Bedeutung:** Auf welche Aussagen wird Prolog mit 'true.' antworten? Was passiert bei der Anfrage '?- es\_regnet.'?



# Deklarative und prozedurale Bedeutung

- Das Ziel bei der Entwicklung von Prolog war eine deklarative Programmiersprache.
- Aber, die deklarative und die prozedurale Bedeutung eines Prologprogramms stimmen nicht immer überein.
- Trotzdem spricht man bei Prolog von einer deklarativen oder logischen Programmiersprache, da sie diesem Ziel nah gekommen ist. (Wer mehr zu dem Thema wissen will, warum es keine bessere Lösung gibt, sollte sich mit dem Problem der Unentscheidbarkeit der Prädikatenlogik befassen.)



# Rekursive Prädikate prozedural

```
teurer(X,Y) :-  
    kostet_etwas_mehr(X,Y).  
  
teurer(X,Y) :-  
    kostet_etwas_mehr(X,Z),  
    teurer(Z,Y).
```

- Erste Regel: Wenn X ist teurer als Y bewiesen werden soll, reicht es X kostet etwas mehr als Y zu beweisen.
- Zweite Regel: Wenn X ist teurer als Y bewiesen werden soll, kann dieses Problem in zwei Teilprobleme zerlegt werden. Gesucht ist ein Z, so dass X etwas mehr kostet als Z (Teilproblem 1) und dass Z teurer ist als Y (Teilproblem 2).



# Übung: rekursive Prädikate

- Wie lauten die Antworten auf die Anfragen? In welcher Reihenfolge werden die Ergebnisse für die letzte Anfrage ausgegeben?

```
verdaut(X,Y) :- hatgegessen(X,Y).  
verdaut(X,Y) :- hatgegessen(X,Z),  
               verdaut(Z,Y).
```

```
hatgegessen(moskito,blut(john)).  
hatgegessen(frosch,moskito).  
hatgegessen(storch,frosch).
```

- 1 ?- verdaut(storch,frosch).
- 2 ?- verdaut(storch,moskito).
- 3 ?- verdaut(frosch,X).
- 4 ?- verdaut(X,Y).



# prozedural verschieden von deklarativ

## Konsequenz für die Prologprogrammierung:

- zunächst sollte man immer das Problem beschreiben (deklarativ),
- anschließend muss man sich Gedanken über die Arbeitsweise (prozedural) des Prolog-Interpreters machen und das Programm gegebenenfalls anpassen.

## Definieren harmloser rekursiver Prädikate:

- Rekursive Prädikate benötigen immer mindestens zwei Klausel: rekursive Klausel plus Anker- oder Ausstiegsklausel.
- Die Ankerklausel sollte immer vor der rekursiven Klausel stehen (sonst droht eine Endlosschleife).
- Im Regelkörper der rekursiven Klausel ist es oft sinnvoll, den rekursiven Aufruf ans Ende zu setzen.



# Beispiel: Definition natürlicher Zahlen

## Natürliche Zahlen

- 0 ist eine natürliche Zahl. (**Ankerregel**)
- Wenn  $n$  eine natürliche Zahl ist, dann ist auch der Nachfolger von  $n$  eine natürliche Zahl. (**rekursive Regel**)
- Nichts sonst ist eine natürliche Zahl. (**Ausschlussregel**)

Wir verwenden `succ/1` zur Kodierung natürlicher Zahlen:

```
0 => 0
1 => succ(0)
2 => succ(succ(0))
3 => succ(succ(succ(0)))
...
...
```

**Ziel:** Ein Prädikat `numeral/1`, welches überprüft ob das Argument eine natürliche Zahl in der `succ`-Darstellung ist.



# Beispiel: Definition natürlicher Zahlen

## Natürliche Zahlen

- 0 ist eine natürliche Zahl. (**Ankerregel**)
- Wenn  $n$  eine natürliche Zahl ist, dann ist auch der Nachfolger von  $n$  eine natürliche Zahl. (**rekursive Regel**)
- Nichts sonst ist eine natürliche Zahl. (**Ausschlussregel**)

**Ziel:** Ein Prädikat `numeral/1`, welches überprüft ob das Argument eine Zahl in der `succ`-Darstellung ist.

```
% Ankerklausel: 0 ist eine Zahl  
numeral(0).
```

```
% rekursive Klausel: Der Nachfolger einer Zahl ist eine Zahl  
numeral(succ(X)) :- numeral(X).
```



# Beispiel: Definition natürlicher Zahlen

Das Programm

```
numeral(0).  
numeral(succ(X)) :- numeral(X).
```

kann zur Generierung von Zahlen genutzt werden:

```
?- numeral(X).  
X = 0 ;  
X = succ(0) ;  
X = succ(succ(0)) ;  
X = succ(succ(succ(0))) ;  
X = succ(succ(succ(succ(0)))) ;  
X = succ(succ(succ(succ(succ(0)))))) ;  
X = succ(succ(succ(succ(succ(succ(0))))))) ;  
X = succ(succ(succ(succ(succ(succ(succ(0)))))))) ;  
X = succ(succ(succ(succ(succ(succ(succ(succ(0)))))))))) ;  
...  
.
```



# Beispiel: Addition natürlicher Zahlen

**Ziel:** Ein Prädikat `add/3`, welches drei Zahlen in der `succ/0`-Darstellung als Argument nimmt.

Das dritte Argument soll die Summe der beiden ersten sein.

```
?- add(succ(0), succ(succ(0)), X).  
X = succ(succ(succ(0))).
```

```
?- add(succ(succ(0)), succ(0), X).  
X = succ(succ(succ(0))).
```

```
?- add(0, succ(succ(0)), X).  
X = succ(succ(0)).
```



# Beispiel: Addition natürlicher Zahlen

**Ziel:** Ein Prädikat `add/3`, welches drei Zahlen in der `succ`-Darstellung als Argument nimmt.

Das dritte Argument soll die Summe der beiden ersten sein.

- **Ankerklausel:** Wenn das erste Argument `0` ist, dann ist das zweite Argument gleich dem dritten Argument.
- **Rekursive Klausel:** Wenn die Summe von `X` und `Y` gleich `Z` ist, dann ist die Summe von `succ(X)` und `Y` gleich `succ(Z)`.

```
% Ankerklausel
add(0, Y, Y).

% rekursive Klausel
add(succ(X), Y, succ(Z)) :-
    add(X, Y, Z).
```



# Beispiel: Addition natürlicher Zahlen

```
% Ankerklausel  
add(0, Y, Y).  
  
% rekursive Klausel  
add(succ(X), Y, succ(Z)):-  
    add(X, Y, Z).
```

Was geschieht bei folgenden Anfragen?

```
?- add(succ(succ(0)) , succ(succ(succ(0))) , Z).  
?- add(X, succ(succ(0)) , succ(succ(succ(0)))).  
?- add(succ(succ(0)) , Y , succ(succ(succ(0)))).  
?- add(X , Y , succ(succ(succ(0)))).  
?- add(X , succ(succ(succ(0))) , Z).  
?- add(succ(succ(succ(0))) , Y , Z).  
?- add(X , Y , Z).
```



# Übung: Addition

Bei der derzeitigen Definition des Prädikats `add/3` erhalten Sie auf manche Anfragen mit mehr als einer Variablen konkrete Zahlen als Antworten, für andere erhalten Sie lediglich eine Angabe über die Beziehungen, die zwischen den Variablenbelegungen herrschen müssen:

```
% keine konkrete Zahl als Antwort
?- add(succ(0),Y,Z).
Z = succ(Y).
```

```
% konkrete Zahlen als Antwort
?- add(X,succ(0),Z).
X = 0,
Z = succ(0) ;
X = succ(0),
Z = succ(succ(0)) ;
X = succ(succ(0)),
Z = succ(succ(succ(0))) ;
...
```

- Können Sie die Definition von `add/3` so anpassen, dass Sie immer konkrete Zahlen als Antwort erhalten?



# Übung: greater\_than

Definieren Sie ein Prädikat `greater_than/2`, das zwei natürliche Zahlen in der `succ/1`-Notation nimmt und überprüft, ob die erste Zahl größer ist als die zweite:

```
?- greater_than(succ(succ(succ(0))), succ(0)).  
true.  
?- greater_than(succ(succ(0)), succ(succ(succ(0)))).  
false.
```



# Zurück zur prozeduralen und deklarativen Bedeutung

**Zur Erinnerung:** Bei der Beweisführung arbeitet sich Prolog

- durch die Wissensbasis von oben nach unten,
- innerhalb der einzelnen Klauseln von links nach rechts durch die Teilziele.

Die Reihenfolge der Klauseln und der Teilziele innerhalb der Klauseln beeinflusst ihre prozedurale Verarbeitung!

```
et(albert,kevin).  
et(lena,albert).  
et(marie,lena).  
  
vorfahr1(X,Y):- et(X,Y).  
vorfahr1(X,Z):-  
    et(X,Y),  
    vorfahr1(Y,Z).  
  
vorfahr2(X,Z):-  
    et(X,Y),  
    vorfahr2(Y,Z).  
vorfahr2(X,Y):- et(X,Y).
```

```
vorfahr3(X,Y):- et(X,Y).  
vorfahr3(X,Z):-  
    vorfahr3(Y,Z),  
    et(X,Y).  
  
vorfahr4(X,Z):-  
    vorfahr4(Y,Z),  
    et(X,Y).  
vorfahr4(X,Y):- et(X,Y).
```

Wie beeinflusst die Reihenfolge das prozedurale Verhalten der Prädikatsdefinitionen?

▶ Übung



# Übung: Vorfahr

Betrachten Sie die folgende Definitionsvariante für das Prädikat **vorfahr**/2. Welche Probleme ergeben sich für diese Variante?

```
vorfahr5(X,Y) :-  
    et(X,Y).
```

```
vorfahr5(X,Y) :-  
    vorfahr5(X,Z),  
    vorfahr5(Z,Y).
```



# Wiederholung: rekursive Prädikate

**Ziele:** Rekursive Prädikate,

- die nicht zu Endlosschleifen führen,
- die möglichst früh terminieren,
- die mit offenen Variablen aufgerufen werden können.

**Definieren harmlöser rekursiver Prädikate:**

- Rekursive Prädikate benötigen immer mindestens zwei Klausel: rekursive Klausel plus Anker- oder Ausstiegsklausel.
- Die Ankerklausel sollte immer vor der rekursiven Klausel stehen (sonst droht Endlosschleife).
- Im Regelkörper der rekursiven Klausel ist es oft sinnvoll, den rekursiven Aufruf ans Ende zu setzen.



# Zusammenfassung

- Wir haben gelernt, dass die Rekursion eine essentielle Programmiertechnik in Prolog ist.
- Wir wissen, dass die Rekursion uns das Schreiben von kompakten und präzisen Programmen ermöglicht.
- Wichtig ist die deklarative sowie prozedurale Bedeutung einer Wissensbasis zu verstehen.
- Keywords: Rekursion, Problemlösungsstrategie, deklarative / prozedurale Bedeutung.
- Wichtig: Die Rekursion ist ein äußerst wichtiges Grundkonzept in Prolog



# Listen in Prolog

- Listen sind sehr mächtige Datenstrukturen in Prolog.
- Listen sind endliche Sequenzen von Elementen:

```
% Liste mit atomaren Elementen:  
[mia, vincent, jules, mia]  
% Liste mit verschiedenen Termen als Elemente:  
[mia, 2, mother(jules), X, 1.7]  
% leere Liste:  
[]  
% Listenelemente koennen Listen sein:  
[mia, [[3,4,paul], mother(jules)], X]
```

- Listen können **verschachtelt** sein (Listen können Listen als Elemente haben)
- Die Reihenfolge der Elemente ist wichtig  $[a,b,c] \neq [b,a,c]$  (Unterschied zu Mengen).
- Dasselbe Element kann mehrfach in einer Liste vorkommen (Unterschied zu Mengen).



# Unifikation / Matching von Listen

Zwei Listen sind unifizierbar,

- wenn sie dieselbe Länge haben und
- wenn die korrespondierenden Listenelemente unifizierbar sind.

```
?- [a,b,X]=[Y,b,3].  
X = 3,  
Y = a  
?- [[a,b,c],b,3]=[Y,b,3].  
Y = [a, b, c]  
?- [a,b,c] = X. % Variablen koennen mit Listen unifiziert werden.  
X=[a,b,c]  
?- [a,b,X,c]=[Y,b,3].  
false.  
?- [a,c,3]=[Y,b,3].  
false.
```

- Die **Länge** einer Lister ist die Zahl ihrer Elemente.



# Listenzerlegung in Prolog

- Prolog hat einen eingebauten Operator ‘|’ (**Listenkonstruktor**) mit dem man eine Liste in **Kopf** (*head*) und **Rest** (*tail*) zerlegen kann.
- Der **Kopf** ist das erste Element der Liste.
- Der **Rest** ist die Restliste.

```
?- [Head|Tail] = [mia, vincent, jules, mia].  
Head = mia,  
Tail = [vincent, jules, mia].
```

- Eine leere Liste hat keinen Head und lässt sich somit nicht zerlegen:

```
?- [Head|Tail] = [].  
false.
```

- Man kann mit ‘|’ auch mehr als ein Element am Anfang abtrennen:

```
?- [First,Second|Tail] = [mia, vincent, jules, mia].  
First = mia,  
Second = vincent,  
Tail = [jules, mia].
```



# Übung: syntaktisch wohlgeformte Listen

- Welche der folgenden Ausdrücke sind syntaktisch wohlgeformte Listen in Prolog?
- Wie lang sind die Listen?

- 1 [1| [2,3,4]]
- 2 [1,2,3| []]
- 3 [1| 2,3,4]
- 4 [1| [2| [3| [4]]]]
- 5 [1,2,3,4| []]
- 6 [[[] | []]]
- 7 [[1,2] | 4]
- 8 [[1,2], [3,4] | [5,6,7]]



# Übung: Matching von Listen

Was antwortet Prolog auf die folgenden Anfragen?

- 1 ?- [a,b,c,d] = [a,[b,c,d]].
- 2 ?- [a,b,c,d] = [a|[b,c,d]].
- 3 ?- [a,b,c,d] = [a,b,[c,d]].
- 4 ?- [a,b,c,d] = [a,b|[c,d]].
- 5 ?- [a,b,c,d] = [a,b,c,[d]].
- 6 ?- [a,b,c,d] = [a,b,c|[d]].
- 7 ?- [a,b,c,d] = [a,b,c,d,[]].
- 8 ?- [a,b,c,d] = [a,b,c,d|[[]]].
- 9 ?- [a,b,c,d] = [a,b,X].
- 10 ?- [a,b,c,d] = [a,b|X].
- 11 ?- [a,b,c,d] = [a,b,[c,d]].
- 12 ?- [a,b,c,d] = [a|[b|[c,d]]].
- 13 ?- [[die,Y)|Z]=[[X,katze],[ist,weg]].
- 14 ?- [a|B]=[A|b]. % Vorsicht: ?- is\_list([a/b]). liefert 'false'.
- 15 ?- [anna,X]=[Y|[maria]].



# Anonyme Variable

- Die Variable ‘`_`’ ist die anonyme Variable in Prolog.
- Sie kommt immer dann zum Einsatz, wenn ein Wert zwar variabel sein soll, später aber nicht mehr benötigt wird.
- Die anonyme Variabel erhöht die Lesbarkeit der Programme.
- Anders als bei anderen Variablen ist jedes Vorkommen der anonymen Variabel unabhängig von den anderen. Sie kann also immer wieder anders initialisiert werden:

```
isst_gerne(X,X) = isst_gerne(a,b).  
false.
```

```
isst_gerne(_,_) = isst_gerne(a,b).  
true.
```

Hinweis: Variablen wie `_X`, die mit einem Unterstrich beginnen sind nicht anonym, sie führen aber im Gegensatz zu anderen Variablen beim Konsultieren einer Wissensbasis nicht zu der Warnung: „**singleton variables**“.



# Anonyme Variable

Beispielproblem: Wir wollen das 2. und 4. Element einer Liste herausgreifen:

```
% ohne anonyme Variabel erhaelt man Werte fuer alle
% Variablen der Anfrage.
?- [X1,X2,X3,X4|T] = [mia, vincent, jules, mia, otto, lena].
X1 = mia,
X2 = vincent,
X3 = jules,
X4 = mia,
T = [otto,lena].  
  
% mit anonymer Variable nur die gesuchten.
?- [_,X2,_,X4|_] = [mia, vincent, jules, mia, otto, lena].
X2 = vincent,
X4 = mia.
```



# Übung: Matching von Listen mit anonymer Variable

Was antwortet Prolog auf die folgenden Anfragen?

```
1 ?- [] = _.
2 ?- [] = [_].
3 ?- [] = [_|[]].
4 ?- [_]=[_|[]].
5 ?- [_,X,_,Y|_] = [dead(zed), [2, [b, chopper]], [], []].
6 ?- [_,X,_,Y|_] = [dead(zed), [2, [b, chopper]], []].
7 ?- [_,_,[_|X]|_] = [[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]].
```



# Prädikat: member/2

- **member/2** ist ein rekursiv definiertes Prädikat, das überprüft, ob ein Element (ein Term) in einer Liste vorkommt:

```
% member/2, member(Term,List)
member(X,[X|_]).           % Base case: if the head of the list is X, then X is a member
member(X,[_|T]) :-           % Recursive case: if the head of the list is not X, then check the rest of the list
    member(X,T).
```

- Wie ist das Programm zu verstehen?



# Prädikat: member/2

- **member/2** ist ein rekursiv definiertes Prädikat, das überprüft, ob ein Element (ein Term) in einer Liste vorkommt:

```
% member/2, member(Term,List)
member(X,[X|_]).           % Fakt
member(X,[_|T]) :-          % Regel
    member(X,T).
```

- Wie ist das Programm zu verstehen?
- Der Fakt `member(X,[X|_]).` besagt, dass etwas ein Element einer Liste ist, wenn es das erste Element (der Head) der Liste ist.
- Die Regel `member(X,[_|T]) :- member(X,T).` besagt, dass etwas ein Element einer Liste ist, wenn es ein Element der Restliste (des Tails) ist.
- Jedes Element einer Liste ist entweder erstes Element oder ein Element im Tail.
- Vorsicht: **member/2** ist ein in der Library *lists* vordefiniertes Prädikat, das von manchen Prologimplementierungen automatisch geladen wird. Verwenden sie daher besser einen anderen Namen, z.B. **my\_member/2**.



# member/2: Beispielanfrage (1)

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

Beispielanfrage:

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

Die erste Klausel passt nicht, aber die zweite. Weiter geht es mit:

```
member(c, [b,c,d]).
```

Und wieder passt nur die rekursive Klausel und es geht weiter mit:

```
member(c, [c,d]).
```

Jetzt passt die erste Klausel (c ist das erste Element der Liste).

Wir bekommen:

```
?- member(c, [a,b,c,d]).  
true.
```



# member/2: Beispielanfrage (2)

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

Beispielanfrage:

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

Die erste Klausel passt nicht, aber die zweite. Weiter geht es mit:

```
member(c, [b]).
```

Und wieder passt nur die rekursive Klausel und es geht weiter mit:

```
member(c, []).
```

Jetzt passt keine der beiden Klauseln, da die Liste leer ist.

Wir bekommen:

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



# Vorteile der deklarativen Programmierung

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

Die deklarative Logik von `member/2` erfasst verschiedene Fälle, für die in prozeduralen Sprachen separate Prozeduren geschrieben werden müssten:

```
% Ist 1 in Liste [1,2,3]?  
?- member(1,[1,2,3]).  
% In welchen Listen ist 1?  
?- member(1,L).  
% Welche X sind in [1,2,3]?  
?- member(X,[1,2,3]).  
% In welchen Listen ist X?  
?- member(X,L).
```

Versuchen Sie Ihre Prädikate immer so zu definieren, dass möglichst alle Anfragerichtungen möglich sind.



# Rekursive Listenverarbeitung an einem Beispiel

Die Definition von Prädikaten, die Listen rekursiv verarbeiten, gehört zu den zentralen Aufgaben in der Prologprogrammierung.

Beispiel: Prädikat **a2b/2**, das zwei Listen L1 und L2 nimmt und genau dann zutrifft, wenn beide Listen gleich lang sind und L1 nur aus a's und L2 nur aus b's besteht.

Vorgehensweise: Zunächst sollte man sich möglichst verschiedene positive und negative Beispiele für die Belegungen der Variablen L1 und L2 überlegen:

```
% positive Beispiele
?- a2b([],[]). % leere Liste
true.
?- a2b([a],[b]).
% Liste mit genau einem Element
true.
?- a2b([a,a],[b,b]).
% Liste mit mehr als einem Element
true.
```



# Rekursive Listenverarbeitung an einem Beispiel

```
% negative Beispiele
?- a2b([a,c,a],[b,b,b]). % L1 besteht nicht nur aus a's
false.
?- a2b([a,a,a],[b,c,b]). % L2 besteht nicht nur aus b's
false.
?- a2b([a,a],[b,b,b]). % L1 ist kuerzer als L2
false.
?- a2b([a,a,a],[b,b]). % L1 ist laenger als L2
false.
?- a2b(t,[b,b]). % L1 ist keine Liste
false.
?- a2b([a,a],t). % L2 ist keine Liste
false.
```



# Rekursive Listenverarbeitung an einem Beispiel

Ausgehend von dieser Aufstellung möglicher Anfragen ist es oft relativ einfach, die Ankerklausel zu formulieren: der Fall mit den einfachsten Listen, die zu einem **true** führen.

```
a2b([], []).
```

Anschließend benötigt man noch die rekursive Klausel: zwei Listen erfüllen die Bedingung des Prädikats **a2b/2**, wenn die erste Liste mit einem a beginnt und die zweite mit einem b und die Restlisten die Bedingung **a2b/2** erfüllen:

```
a2b([], []).
```

```
a2b([a|T1], [b|T2]) :-  
    a2b(T1, T2).
```

Abschließend sollte man immer die Prädikatsdefinition mit den zuvor aufgestellten Positiv- und Negativbeispielen testen.



# Übung: auf member/2 beruhende Prädikate

Für die Definition der folgenden Prädikate kann `member/2` verwendet werden.

- 1 Schreibe ein Prädikat `all_members/2`, das zwei Listen L1 und L2 nimmt und gelingt, wenn alle Elemente von L1 auch Element von L2 sind.

```
?- all_members([a,c],[a,b,c,d]).  
true.  
?- all_members([a,e],[a,b,c,d]).  
false.  
?- all_members(a,[a,b,c,d]).  
false.
```

- 2 Schreibe ein Prädikat `set_equal/2`, das zwei Listen L1 und L2 nimmt und gelingt, wenn die beiden Listen als Mengen betrachtet gleich sind (also die gleichen Elemente haben).

```
?- set_equal([a,b,a],[b,b,b,a]).  
true.  
?- set_equal([a,b,c],[b,b,b,a]).  
false.  
?- set_equal([a,b],[c,a,b]).  
false.
```



# Interne rekursive Listenrepräsentation

Prolog behandelt nichtleere Listen intern als zweistellige zusammengesetzte Terme mit Funktor `'[]'`.

```
?- [a,b] = '[]'(a,'[]'(b,[])).  
true.
```

Nichtleere Listen werden dabei in Kopf und Rest zerlegt.

Der Rest ist entweder die leere oder wiederum eine nichtleere Liste.

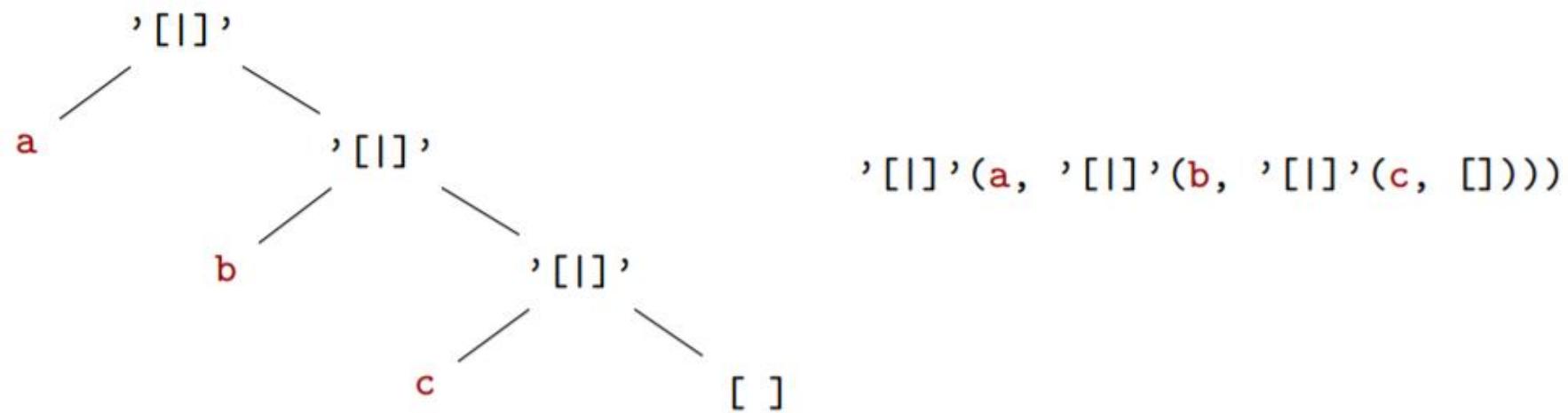
<code>'[]'(a, [])</code>	<code>[a   []]</code>	<code>[a]</code>
<code>'[]'(a, '[]'(b, []))</code>	<code>[a   [b   []]]</code>	<code>[a,b]</code>
<code>'[]'(a, '[]'('[]'(c, [])))</code>	<code>[a   [b   [c   []]]]</code>	<code>[a,b,c]</code>

Statt `'[]'` verwenden viele Implementierungen ein anderes Symbol (z.B. `'.'`).



# Interne rekursive Listenrepräsentation

Listen können als binäre Bäume aufgefasst werden:



# Übung: einfache Listenprädikate

- Schreiben sie ein Prädikat `third/2`, das gelingt, wenn das zweite Argument eine Liste ist und das erste Argument das dritte Element dieser Liste ist.

```
?- third(a,[b,c,a,d,e]).  
true.
```

- Schreiben sie ein Prädikat `tausch12/2`, das zwei Listen als Argumente nimmt und gelingt, wenn sich die beiden Listen nur in der Reihenfolge der ersten beiden Elemente unterscheiden.

```
?- tausch([a,b,c,d],[b,a,c,d]).  
true.
```



# Übung: Beweis und Suchbaum

Gegeben ist folgende Wissensbasis:

```
house_elf(dobby).  
witch(hermione).  
witch(mcGonagall).  
witch(rita_skeeter).  
wizard(goofy).  
magic(X):-house_elf(X).  
magic(X):-wizard(X).  
magic(X):-witch(X).
```

Welche der folgenden Anfragen lassen sich beweisen, wie werden eventuelle Variablen belegt?

- 1 ?- magic(house\_elf).
- 2 ?- wizard(harry).
- 3 ?- magic(wizard).
- 4 ?- magic(mcGonagall).
- 5 ?- magic(Hermione).

Zeichne den Suchbaum für die 4. Anfrage. Gib alle Lösungen für die 5. Anfrage in der Reihenfolge an, in der sie Prolog ausgeben würde.



# Übung: Anfragen

- Wie lauten die Antworten auf die Anfragen? In welcher Reihenfolge werden die Ergebnisse für die letzte Anfrage ausgegeben?

```
verdaut(X,Y) :- hatgegessen(X,Y).  
verdaut(X,Y) :- hatgegessen(X,Z),  
               verdaut(Z,Y).
```

```
hatgegessen(moskito,blut(john)).  
hatgegessen(frosch,moskito).  
hatgegessen(storch,frosch).
```

- 1 ?- **verdaut(storch,frosch).**
- 2 ?- **verdaut(storch,moskito).**
- 3 ?- **verdaut(frosch,X).**
- 4 ?- **verdaut(X,Y).**



# Übung: Listen

- Definiere ein Prädikat *vergleich/2*, das zwei Listen L1 und L2 nimmt und genau dann zutrifft, wenn beide Listen gleich lang sind und L1 nur aus a's und L2 nur aus b's besteht.



# Übung: Listen

- Welche der folgenden Ausdrücke sind syntaktisch wohlgeformte Listen in Prolog?
- Wie lang sind die Listen?

- 1 [1| [2,3,4]]
- 2 [1,2,3| []]
- 3 [1| 2,3,4]
- 4 [1| [2| [3| [4]]]]
- 5 [1,2,3,4| []]
- 6 [[[] | []]]
- 7 [[1,2] | 4]
- 8 [[1,2], [3,4] | [5,6,7]]



# Übung: Listen

Was antwortet Prolog auf die folgenden Anfragen?

```
1 ?- [a,b,c,d] = [a,[b,c,d]] .  
2 ?- [a,b,c,d] = [a|[b,c,d]] .  
3 ?- [a,b,c,d] = [a,b,[c,d]] .  
4 ?- [a,b,c,d] = [a,b|[c,d]] .  
5 ?- [a,b,c,d] = [a,b,c,[d]] .  
6 ?- [a,b,c,d] = [a,b,c|[d]] .  
7 ?- [a,b,c,d] = [a,b,c,d,[]] .  
8 ?- [a,b,c,d] = [a,b,c,d|[[]]] .  
9 ?- [a,b,c,d] = [a,b,X] .  
10 ?- [a,b,c,d] = [a,b|X] .  
11 ?- [a,b,c,d] = [a,b,[c,d]] .  
12 ?- [a,b,c,d] = [a|[b|[c,d]]] .  
13 ?- [[die,Y)|Z]=[[X,katze],[ist,weg]] .  
14 ?- [a|B]=[A|b] . % Vorsicht: ?- is_list([a/b]). liefert 'false'.  
15 ?- [anna,X]=[Y|[maria]] .
```



# Übung: Listen

Was antwortet Prolog auf die folgenden Anfragen?

```
1 ?- [] = _.
2 ?- [] = [_].
3 ?- [] = [_|[]].
4 ?- [_]=[_|[]].
5 ?- [_,X,_,Y|_] = [dead(zed), [2, [b, chopper]], [], []].
6 ?- [_,X,_,Y|_] = [dead(zed), [2, [b, chopper]], []].
7 ?- [_,_,[_|X]|_] = [[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]].
```



# Übung: Suchbäume

Zeichnen sie die Suchbäume zu den folgenden Anfragen:

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

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

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



# Übung: Listen

Für die Definition der folgenden Prädikate kann `member/2` verwendet werden.

- 1 Schreibe ein Prädikat `all_members/2`, das zwei Listen L1 und L2 nimmt und gelingt, wenn alle Elemente von L1 auch Element von L2 sind.

```
?- all_members([a,c],[a,b,c,d]).  
true.  
?- all_members([a,e],[a,b,c,d]).  
false.  
?- all_members(a,[a,b,c,d]).  
false.
```

- 2 Schreibe ein Prädikat `set_equal/2`, das zwei Listen L1 und L2 nimmt und gelingt, wenn die beiden Listen als Mengen betrachtet gleich sind (also die gleichen Elemente haben).

```
?- set_equal([a,b,a],[b,b,b,a]).  
true.  
?- set_equal([a,b,c],[b,b,b,a]).  
false.
```



# Übung: Listen

- Definiere ein rekursives Prädikat *länge/2*, welches die Länge einer Input Liste zurückgibt.

