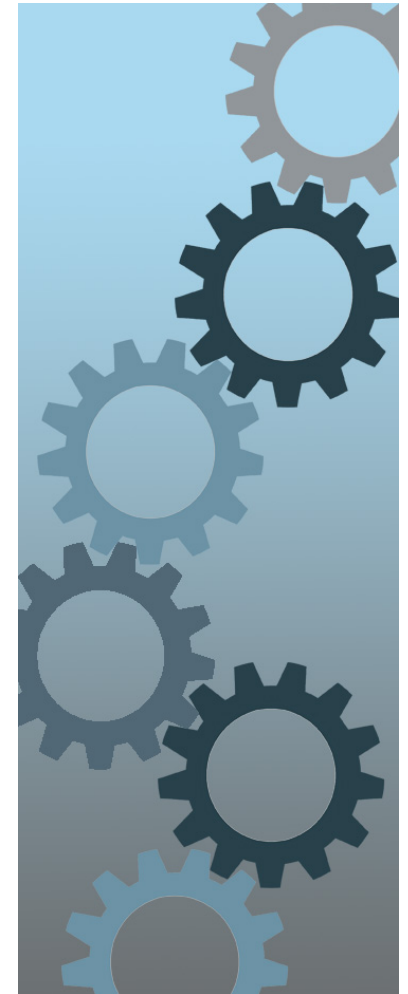


Programming Concepts & Paradigms

Prolog 3

Prof. Dr. Ruedi Arnold

ruedi.arnold@hslu.ch



Übersicht Prolog 3

- Optimierungen
 - Endrekursion
 - Memoization
- Datenstruktur Listen
 - Syntax & rekursiver Aufbau
 - Zwei Operationen zur Anwendung: Zugehörigkeit (mem/2) und Konkatenation (conc/3)

Wiederholung SW01:
alles ok? Fragen, Unklarheiten?

Zulassung und Nachweis

- Zulassung (Testat)
 - Min. 5 Wochenübungen (von 9) zufriedenstellend gelöst und einem Dozenten oder Assistenten individuell präsentiert, davon zwingend Übung SW4 und SW7
 - 2 Ilias-Übungen gelöst (je 1 zu Prolog & Scheme)
- Nachweis **(neu und anders als im Modulbescrieb!)**
 - **1/3:** Aktive Teilnahme an einem individuellen 2er-Team-Projekt, inkl. Präsentation und Diskussion der erarbeiteten Resultate im Plenum.
 - **2/3:** Schriftliche Modulendprüfung, **2h** ~~(3h)~~



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Optimierung durch Endrekursion

Programm-Optimierungen

- Gesehen: Prolog verwendet grundsätzlich Backtracking, d.h. Tiefensuche zur Problemlösung
 - Backtracking ist grundsätzlich nicht sehr effizient
 - Zwei bekannte Methoden zur Effizienzoptimierung
 - Endrekursion
 - Assertions
- schauen wir nun beide am Beispiel Fibonacci an

Endrekursion (tail recursion)

- Eine Prozedur ist endrekursiv, wenn
 - sie nur einen rekursiven Aufruf hat und
 - dieser rekursive Aufruf ist der letzte Aufruf in der letzten Klausel von dieser Prozedur
 - Zusätzlich müssen die Aufrufe vor dem rekursiven Aufruf alle deterministisch sein
- Vorteil: Kein Backtracking notwendig!
 - Endrekursion kann als Iteration ohne zusätzlichen Speicherplatz ausgeführt werden

Endrekursion: allg. Beispiel

```
p(...) :- ...           % no recursive call in the body
p(...) :- ...           % no recursive call in the body
...
p(...) :- ...,          % all deterministic and
                    ..., % no recursive calls until here.
                    p(...). % here: tail-recursive call
```

- Eine derartige endrekursive Prozedur wird in Prolog als Iteration ausgeführt
 - Braucht viel weniger Speicherplatz als rekursive Aufrufe (und ggf. Backtracking)
- Hinweis: Diese Optimierung heisst auch „last call optimization“

Fibonacci-Zahlen: naives rekursives Programm

```
▪ fib(0, 0).  
  fib(1, 1).  
  fib(N, F) :-  
    N > 1,  
    N1 is N - 1, N2 is N - 2,  
    fib(N1, F1), fib(N2, F2),  
    F is F1 + F2.
```

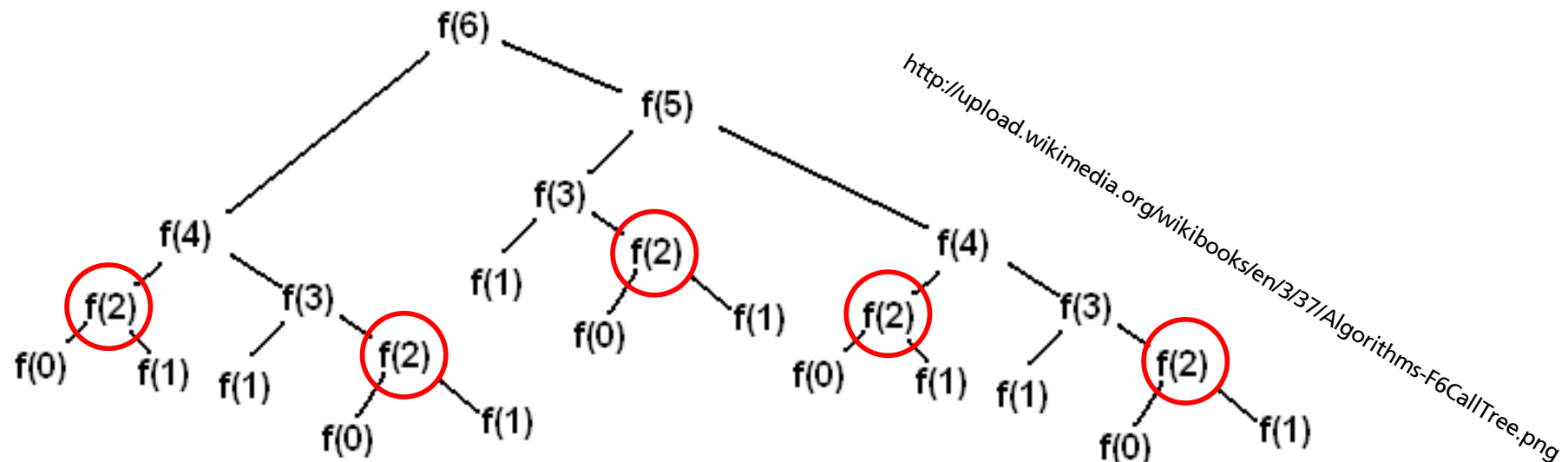
- Problem: Rekursive Berechnung ist nicht sehr effizient
 - Grund: Viele geschachtelte rekursive Aufrufe benötigen viel Speicherplatz! Führt auf meinem MacBook dazu:

```
?- fib(30,X).  
ERROR: Out of local stack
```

- Aufruf von fib(27,X) braucht ca. 10 Sekunden...

Fibonacci rekursiv: Aufrufbaum & Ineffizienz...

- Aufrufbaum zur siebten Fibonacci-Zahl $f(6)$



- Beobachtung: Viele Zwischenresultate werden aufgrund der rekursiven Aufrufbaum-Struktur mehrmals berechnet. Zeitkomplexität $O(2^n)$
 - $f(2)$ wird im Beispiel oben z.B. bereits **5x** berechnet!

Fibonacci-Zahlen mittels Endrekursion

```
fib_tr(N, F) :- fib_tr(N, 0, 1, F). % call accumulator
fib_tr(0, A, _, A).                % simple case
fib_tr(N, A, B, F) :-              % general case
    N1 is N - 1,                  % new argument N1
    N1 >= 0,                      % avoid underflow
    Sum is A + B,                 % accumulator Sum
    fib_tr(N1, B, Sum, F).        % tail-recursive call
```

- Hinweis: Umwandlung von allg. Prolog-Programm in endrekursives ist i.A. nicht trivial (und nicht Prüfungs-relevant)
 - Typischerweise werden dabei zusätzliche neue Argumente verwendet: Akkumulatoren. Diese werden verwendet, um Zwischenwerte zu speichern und so inkrementell das Schlussresultat zu berechnen
 - Mehr zu akkumulativer Rekursion im Scheme-Teil von PCP

Performanz endrekursiver Fibonacci-Zahlen

- Ohne Verzögerung (auf meinem MacBook Pro)

```
?- fib_tr(30, X) .
```

```
X = 832040 .
```

```
?- fib_tr(100, X) .
```

```
X = 354224848179261915075 .
```

```
?- fib_tr(1000,X) .
```

```
X = 434665576869374564356885276750406258025646605173717804  
0248172908953655541794905189040387984007925516929592259308  
0322634775209689623239873322471161642996440906533187938298  
969649928516003704476137795166849228875 .
```

→ d.h.: beeindruckende Optimierung vom endrekursiven fib_tr/2-Prädikat gegenüber dem naiven fib/2

Fazit Endrekursion

- Falls der Speicherplatz bei einer rekursiven Prozedur kritisch ist, hilft die Umwandlung in eine endrekursive Prozedur
 - Eine endrekursive Prozedur kann von Prolog als Iteration ausgeführt werden
 - Umwandlung von rekursiver in endrekursive Prozedur ist i.A. nicht trivial und benötigt meistens ein Akkumulator-Argument (oder ggf. mehrere)

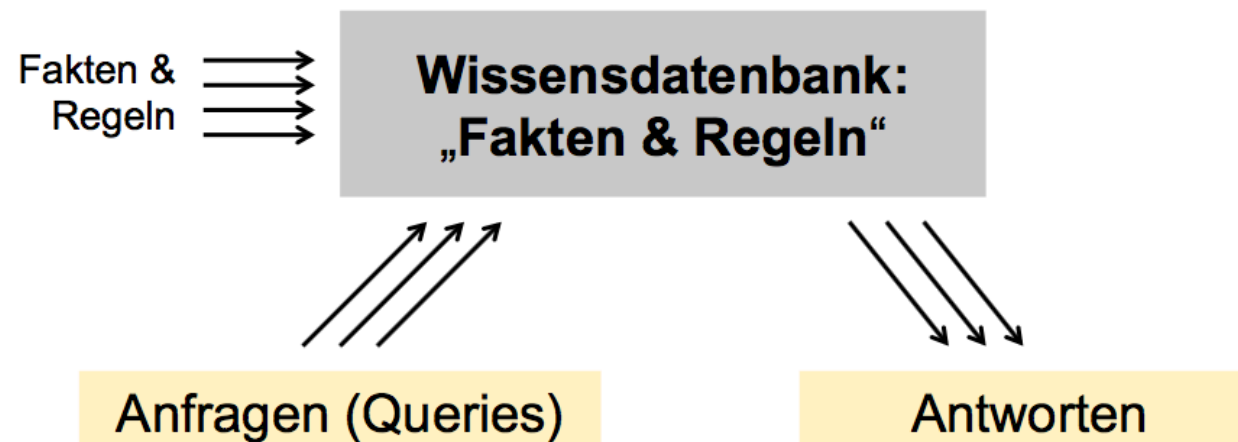


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Optimierung durch Assertions: Memoization

Memoization: Optimierung durch Assertions

- Die Wissensdatenbank von Prolog kann durch Programme manipuliert werden
 - D.h. Fakten und Regeln können dynamisch hinzugefügt oder gelöscht werden
- Dies kann dazu genutzt werden, um Programme zu optimieren. Fachbegriff: Memoization (auch: Caching)
 - Schauen wir später am Beispiel Fibonacci an



Fakten/Regeln anzeigen: listing/1

- Beispiel-Fakten (siehe Prolog 1)

```
:- dynamic bigger/2.  
bigger(elephant, horse).  
bigger(horse, dog).  
bigger(horse, sheep).
```

- Hinweis: Anweisung `dynamic/1` ist notwendig, damit das statische (= aus einer Datei geladene) Prädikat `bigger/2` zur Laufzeit in SWI-Prolog modifiziert werden darf

- Fakten/Regeln zu `bigger/2` anzeigen lassen: `listing/1`

```
?- listing(bigger).  
:- dynamic bigger/2.  
bigger(elephant, horse).  
bigger(horse, dog).  
bigger(horse, sheep).
```

Fakten/Regeln hinzufügen: asserta/1

- Neuen Fakt hinzufügen und anzeigen lassen

```
?- asserta(bigger(me, you)) .  
true.
```

```
?- listing(bigger) .  
:- dynamic bigger/2.
```

```
bigger(me, you) .                                % new fact as first rule  
bigger(elephant, horse) .  
bigger(horse, dog) .  
bigger(horse, sheep) .
```

- So lassen sich Programme zur Laufzeit modifizieren!

Einfüge-Ort: asserta/1 vs. assertz/1

- Wo soll neuer Fakt / neue Regel eingefügt werden?
 - Erste Regel / erster Fakt: asserta/1
 - Letzte Regel / letzter Fakt: assertz/1
- Beispiel (weitergeführt von vorgehender Folie...)

```
?- assertz(bigger(elephant, me)) .
true.

?- listing(bigger) .
:- dynamic bigger/2.

bigger(me, you) .
bigger(elephant, horse) .
bigger(horse, dog) .
bigger(horse, sheep) .
bigger(elephant, me) .           % new fact asserted as last rule
```

Fakten/Regeln entfernen: retract/1

- Fakten und Regeln können aus der Wissensdatenbank entfernt werden mittels retract/1
- Beispiel (weitergeführt von vorgehender Folie...)

```
?- retract(bigger(me, you)).  
true.  
  
?- listing(bigger).  
:- dynamic bigger/2.  
  
bigger(elephant, horse).  
bigger(horse, dog).  
bigger(horse, sheep).  
bigger(elephant, me).
```

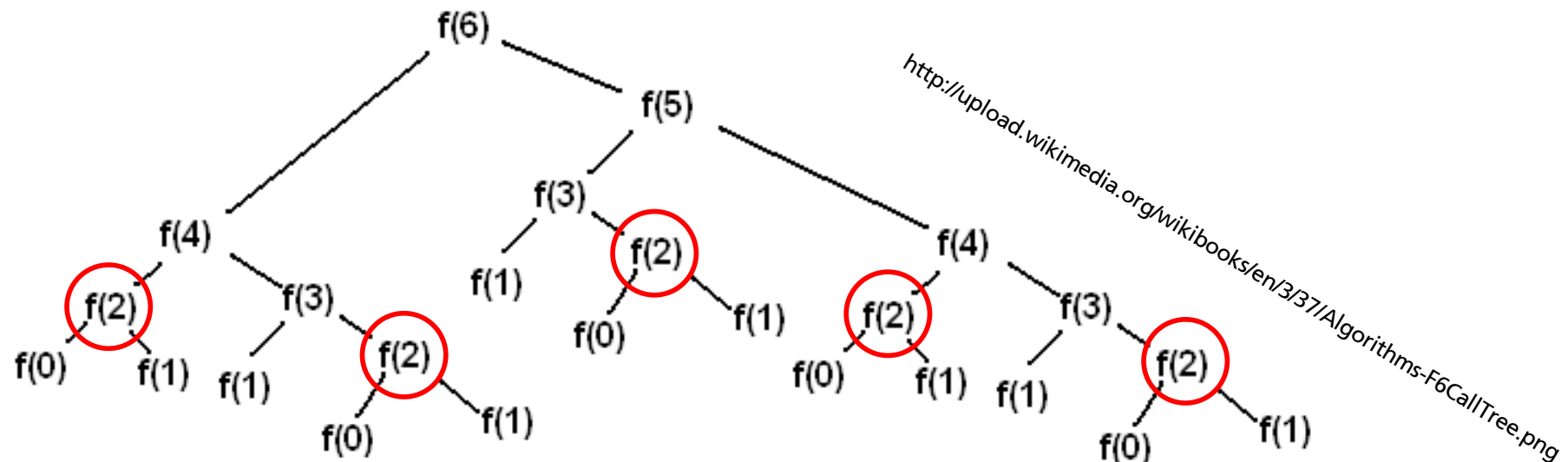
– damit ist der Fakt bigger(me, you) wieder gelöscht

Mit Assertions optimieren

- Assertions können eingesetzt werden, um Programme zu optimieren
 - Typischer Fall: Gewisse Teilprobleme müssen nicht mehrmals gelöst werden, sondern die Lösungen werden mit Hilfe von Assertions in der Wissensdatenbank abgelegt
- Beispiel: Rekursive Berechnung der Fibonacci-Zahlen
 - Beobachtung: Viele Zwischenresultate werden aufgrund der rekursiven Aufrufbaum-Struktur mehrmals berechnet
 - Das ist ineffizient, da mehrmals genau dasselbe berechnet wird 😞

Fibonacci rekursiv: Aufrufbaum & Ineffizienz...

- Aufrufbaum zur siebten Fibonacci-Zahl $f(6)$



- Beobachtung: Viele Zwischenresultate werden aufgrund der rekursiven Aufrufbaum-Struktur mehrmals berechnet. Zeitkomplexität $O(2^n)$
 - $f(2)$ wird im Beispiel oben z.B. bereits **5x** berechnet!

Fibonacci optimiert mit Assertions

```
:- dynamic fib_as/2.  
fib_as(0, 0).           % base case 1  
fib_as(1, 1).           % base case 2  
fib_as(N, F) :-         % general rule  
    N > 1,              % allow no negative numbers  
    N1 is N-1,  
    N2 is N-2,  
    fib_as(N1, F1),      % calculate F1 = fib(N-1)  
    fib_as(N2, F2),      % calculate F2 = fib(N-2)  
    F is F1+F2,  
    asserta(fib_as(N, F)). % assert new fact
```

- Implementation analog zu naivem Fibonacci-Code, neu werden bereits berechnete Fibonacci-Zahlen mittels `asserta/1` zur Wissensdatenbank hinzugefügt

Performanz Fibonacci-Zahlen mit Assertions

- Ohne Verzögerung (auf meinem MacBook)

```
?- fib_as(30, X).
```

```
X = 832040 .
```

```
?- fib_as(100, X).
```

```
X = 354224848179261915075 .
```

```
?- fib_as(1000,X).
```

```
X = 434665576869374564356885276750406258025646605173717804  
0248172908953655541794905189040387984007925516929592259308  
0322634775209689623239873322471161642996440906533187938298  
969649928516003704476137795166849228875 .
```

→ d.h.: starke Optimierung durch die Verwendung von Assertions bei fib_as/2 gegenüber dem naiven fib/2

– Zur Erinnerung:

```
?- fib(30,X).
```

```
ERROR: Out of local stack
```

Kontrollfragen A

1. Warum kann eine endrekursive Prozedur von Prolog effizienter ausgeführt werden, als eine nicht-endrekursive Prozedur?
2. Beschreiben sie in eigenen Worten, auf was für einer Beobachtung „Optimierung mit Assertions“ beruht und wie diese konkret in Prolog umgesetzt wird.



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Listen: Syntax & Aufbau

Datenstruktur Liste

- Eine Liste ist eine endliche Sequenz von Elementen
- Listen können in Prolog mit Hilfe von [] (eckigen Klammern) dargestellt werden, z.B.:

```
?- X = [a, b, c].
```

```
X = [a, b, c].
```

```
?- Y = [d, e, f(X), [x, y]].
```

```
Y = [d, e, f(X), [x, y]].
```

```
?- Z = [].
```

```
Z = [].
```

Eigenschaften von Listen & die leere Liste []

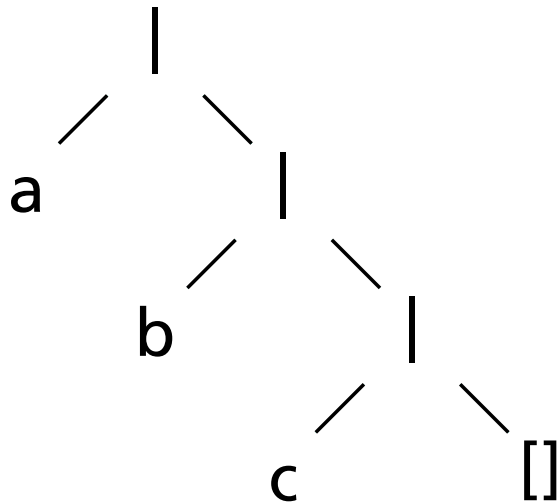
- Die Elemente einer Liste werden in eckigen Klammern eingeschlossen und durch Komma getrennt
- Die Länge einer Liste ist die Anzahl Elemente, welche in dieser Liste enthalten sind
- Listen-Elemente sind beliebige Prolog Terme
 - Also z.B. Atome, Zahlen oder wiederum Listen
- Es gibt eine spezielle Liste, nämlich die leere Liste, angegeben als []

Wie sind Listen aufgebaut?

- Eine nicht-leere Liste besteht immer aus zwei Dingen:
 1. Einem ersten Element = **Kopf** der Liste (head)
 2. Dem restlichen Teil der Liste = **Schwanz** der Liste (tail)
 - Der Schwanz einer Liste ist immer auch wieder eine Liste

 - Von der Liste [a, b, c] ist z.B.
 - a der Kopf der Liste
 - und die Liste [b, c] ist der Schwanz der Liste
- Listen sind also rekursiv aufgebaut!

Baumdarstellung der Liste [a, b, c]



- Jede Liste kann rekursiv als Baum dargestellt werden
 - Linke Kinder: Elemente, rechte Kinder: „Rest-Liste“
 - Wichtig: Terminierung durch [], d.h. durch einer leere Liste. (Dies entspricht praktisch der Rekursionsbasis)

Listen-Operator | und Anwendung [Head | Tail]

- Mit der |-Notation (senkrechter Strich, EN: Pipe) kann eine Liste in Kopf (head) und Schwanz (tail) unterteilt werden, z.B.:

```
?- [a, b, c] = [Head | Tail].  
Head = a,  
Tail = [b, c].
```

- Diese Notation kann auch allgemeiner eingesetzt werden, indem wir eine beliebige Anzahl von Elementen einer Liste, gefolgt von | und einer Liste mit den restlichen Elementen angeben. Beispiel:

```
?- L = [a | [b, c]], L = [a, b | [c]], L = [a, b, c | []].  
L = [a, b, c].
```

Beispiel-Einsatz von [Head | Tail]: n-tes Element

- Angenommen, wir sind am zweiten und dritten Element einer Liste interessiert, diese erhalten wir mit Hilfe von | (senkrechter Strich, pipe-Operator) z.B. so:

```
?- [_ , X2 , X3 | _] = [a , b , c , d , e , f] .  
X2 = b ,  
X3 = c .
```

- Hinweis: Für das erste Element und den Rest der Liste nach dem dritten Element verwenden wir hier die anonyme Variable `_`, da uns diese beiden Dinge nicht weiter interessieren

Kontrollfragen B

1. Was antwortet Prolog auf die folgende Anfrage:
 $X = [a \mid [b]]?$
2. Was antwortet Prolog auf die folgende Anfrage:
 $[1, 2, 3] = [_ \mid X]?$
3. Und was auf diese: $[a, b, c] = [_, X \mid Y]?$
4. Warum ist es in Prolog i.A. effizienter auf das erste, als auf das letzte Element einer Liste zuzugreifen?



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

Listenoperationen

Listenzugehörigkeit (list membership)

- Wir wollen wissen, ob ein Element X in einer Liste L vorkommt oder nicht
- Wir definieren dazu eine Relation $\text{mem}(X, L)$
 - Für diese Relation soll z.B. gelten
 - $\text{mem}(b, [a, b, c])$ stimmt
 - $\text{mem}(b, [a, [b, c]])$ stimmt nicht
 - $\text{mem}(d, [a, b, c])$ stimmt nicht
 - $\text{mem}([b, c], [a, [b, c]])$ stimmt
- Wie gehen wir dazu vor?

Listenzugehörigkeit: Prädikat `mem(X, L)`

- Element `X` kommt in Liste `L` vor wenn entweder
 - a) `X` der Kopf ist von `L` oder
 - b) `X` kommt im Schwanz von `L` vor
- Diese rekursive Beobachtung kann direkt in die folgenden zwei Klauseln umgeschrieben werden

```
mem(X, [X | _]).                % tail doesn't matter  
mem(X, [_ | Tail]) :- mem(X, Tail). % head doesn't matter
```

- Hinweis: Klauseln verwenden anonyme Variablen
 - Falls nicht: Prolog-Warnung „**Singleton Variables**“

Anwendungen von mem/2

- „Normale“ Benutzung, d.h. Testen, ob ein Element in einer Liste vorkommt

```
?- mem(a, [a, b, c]).           % is a member of [a, b, c]?  
true.  
  
?- mem(d, [a, b, c]).           % is d member of [a, b, c]?  
false.  
  
?- mem([b, c], [a, [b, c]]).    % is [b, c] member of [a, [b, c]]?  
true.
```

- Generierung aller Elemente einer Liste

```
?- mem(X, [a, b, c]).           % which X is a member of [a, b, c]?  
X = a;  
X = b;  
X = c;  
false.
```

Weitere Anwendungen von mem/2

- Finde eine Liste L, in der hslu vorkommt

```
?- mem(hslu, L). % of which list L is hslu a member of?
L = [hslu|_G2031];
L = [_G2030, hslu|_G2034];
L = [_G2030, _G2033, hslu|_G2037];
L = [_G2030, _G2033, _G2036, hslu|_G2040];
...
```

- Finde Listen, die a, b, und c enthalten

```
?- mem(a, L), mem(b, L), mem(c, L). % which list L contains a, b, & c?
L = [a, b, c|_G2239];
L = [a, b, _G2238, c|_G2242];
L = [a, b, _G2238, _G2241, c|_G2245];
...
```

Erzeugung von Permutationen mittels mem/2

- Bei den letzten beiden Beispielen war die Reihenfolge der Elemente vorgegeben. Prolog generiert (unendlich) lange Lösungslisten, aber keine Permutationen. Grund: Backtracking ist Tiefensuche
- Frage: Wie können wir Permutationen erzeugen?
- Antwort: Wir beschränken die Länge der Liste!

```
?- L = [_ , _ , _] , mem(a, L) , mem(b, L) , mem(c, L) .  
L = [a, b, c] ;  
L = [a, c, b] ;  
L = [b, a, c] ;  
L = [c, a, b] ;  
L = [b, c, a] ;  
L = [c, b, a] ;  
false.
```

– Erzeugt alle $3! = 6$ Permutationen von a, b und c 😊

Bemerkungen zu mem/2 und member/2

- Ein zu mem/2 zum Testen auf Listenzugehörigkeit ist bereits in Prolog eingebaut: member/2
 - D.h. wir können das member/2-Prädikat verwenden
- Für versierte Prolog-Programmier*innen ist es jedoch wichtig zu verstehen, wie Listen-Prädikate funktionieren und wie diese selber geschrieben werden können
 - Eine weiteres Listen-Prädikat (Konkatenation) schauen wir jetzt noch in der Vorlesungen an, weitere tauchen dann in den Übungen auf
 - Typischerweise funktionieren diese alle nach dem gleichen rekursiven Ansatz: Wie sieht die Lösung für den einfachen Fall aus, und wie für den allgemeinen Fall?

Listenkonkatenation (list concatenation)

- Wir wollen wissen, ob Liste L1 und Liste L2 zusammenhängt Liste L3 ergeben
- Wir definieren dazu eine Relation $\text{conc}(L1, L2, L3)$
 - Für diese Relation soll z.B. gelten
 - $\text{conc}([a, b], [c, d], [a, b, c, d])$ stimmt
 - $\text{conc}([a, b], [c], [a, b, c, d])$ stimmt nicht
 - $\text{conc}([a, b, c], [d], [a, b, c, d])$ stimmt
 - $\text{conc}([a, b, c], [], [c, b, a])$ stimmt nicht

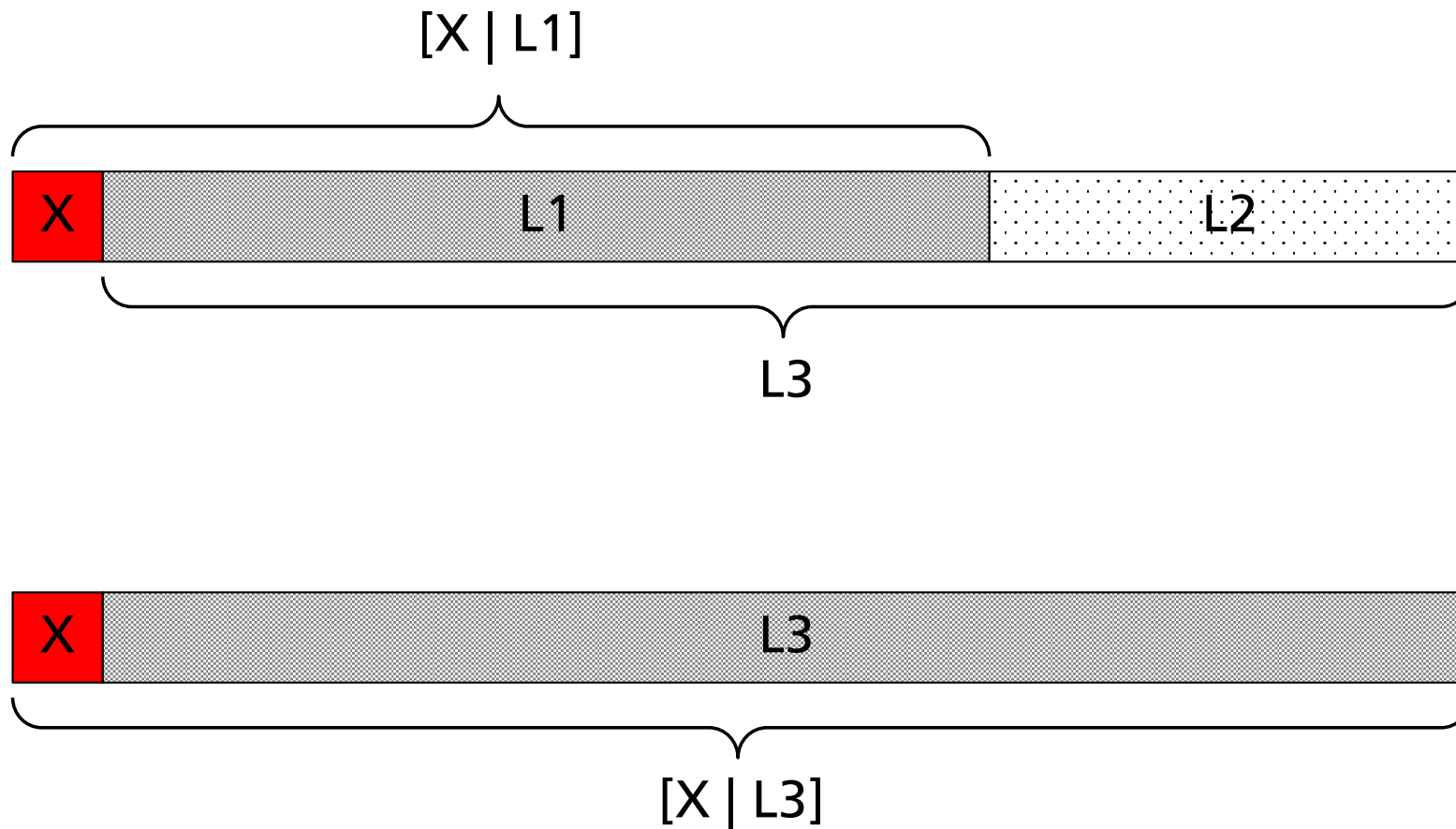
Listenkonkatenation : Prädikat conc/3

- Analog zu mem/2 unterscheiden wir auch hier wieder zwei Fälle, abhängig von L1:
 - a) Wenn die erste Liste die leere Liste ist, dann müssen die zweite und die dritte dieselbe Liste sein
 - b) Wenn die erste Liste nicht leer ist, dann hat sie einen Kopf und einen Schwanz und sieht so aus: $[X \mid L1]$. Das Resultat der Konkatenation ist die Liste $[X \mid L3]$, wobei L3 die Konkatenation der Listen L1 und L2 ist
- Diese zwei beobachteten Fälle in Prolog

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


Illustration vom allg. Fall von $\text{conc}(\text{L1}, \text{L2}, \text{L3})$

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



Anwendungen von conc/2

- „Normale“ Benutzung, d.h. zwei Listen konkatenieren

```
?- conc([a, b], [c, d], L).  
L = [a, b, c, d].
```

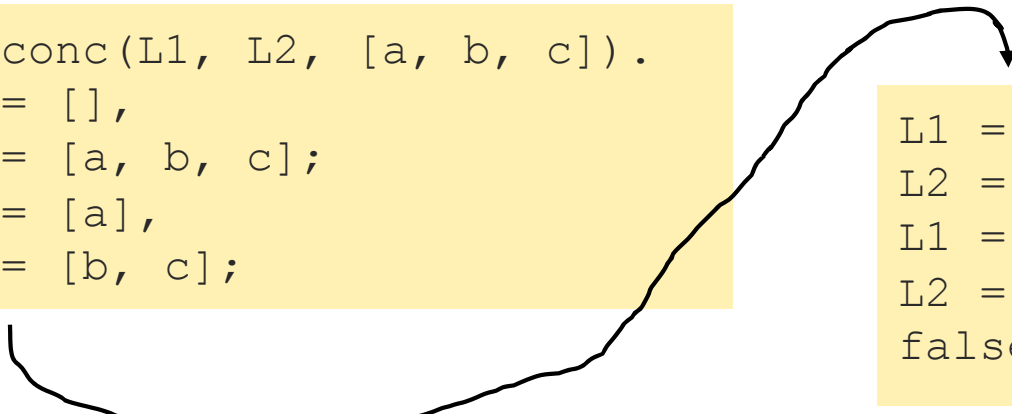
```
?- conc([a, b], [c], [a, b, c, d]) .  
false.
```

```
?- conc([a, [b, c], d], [a, [], b], L).  
L = [a, [b, c], d, a, [], b].
```

- Zerlegung einer Liste in alle möglichen Teillisten

```
?- conc(L1, L2, [a, b, c]).  
L1 = [],  
L2 = [a, b, c];  
L1 = [a],  
L2 = [b, c];
```

```
L1 = [a, b],  
L2 = [c];  
L1 = [a, b, c],  
L2 = [];  
false.
```



Kontrollfragen C

`conc/3` sei wie oben beschrieben definiert.

1. Was antwortet Prolog auf die folgende Anfrage:

`conc(L, [c], [a, b, c])?`

2. Was antwortet Prolog auf die Anfrage

`conc(Before, [d | After], [a, b, c, d, e, f, g, h])?`

3. Und was auf `conc([a], L, [b, c])?`

4. Bonusaufgabe (anspruchsvoll): Wie lässt sich `mem/2` unter Verwendung von `conc/3` ausdrücken? (D.h. als neues Prädikat der Form `mem_c(X, L) :- ...conc...`)