Prolog 3

Wie optimiert man Prolog-Programme?

- Prolog verwendet grundsätzlich Backtracking, d.h. Tiefensuche zur Problemlösung
- grundsätzlich nicht effizient
- Zwei Methoden zur Optimierung
    - Endrekursion
    - Assertions

Was ist Endrekursion?

- rekursive Funktion, bei der
- der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von *f* ist

Wann ist eine Prozedur endrekursiv?

Wenn,

- sie nur einen rekursiven Aufruf hat und
- dieser rekursive Aufruf der letzte Aufruf in der letzten Klausel von dieser Prozedur ist
- Aufrufe vor dem rekursiven Aufruf alle deterministisch sind

> Alle Bedingungen müssen erfüllt sein

Was ist der Vorteil von Endrekursion?

- kein zusätzlicher Speicherplatz zur Verwaltung der Rekursion notwendig
- kein Backtracking notwendig
- Endrekursion kann als Iteration ohne zusätzlichen Speicherplatz ausgeführt werden

Beispiel Endrekursion

```
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
```

> Iteration (weniger Speicher benötigt)

Was ist mit "last call optimization" gemeint?

- Endrekursion

Was ist mit Memoization gemeint?

- Zwischenspeichern der (Zwischen-)Resultate von ineffizienten Funktionsaufrufen,
- um diese Resulte später wieder aus dem Cache holen zu können bei gleichen Funktionsaufrufen

Wofür verwendet man `dynamic` ?

- Um statische (= aus einer Datei geladene) Prädikate zur Laufzeit modifizieren zu können
- Brauchen wir bei Memoization (Assertion) um neue Fakten und Regeln zur Laufzeit hinzufügen zu können

Wie kann man zur Laufzeit Fakten / Regeln anzeigen lassen?

- Mit dem Prädikat `listing(<Prädikat hier einfügen>)`

Wie fügt man neue Fakten / Regeln einem Prädikat zur Laufzeit hinzu?

- Mit dem Prädikat `asserta(<Regel / Fakt hier einfügen>)`
- Diese neue Regeln wird zur Laufzeit hinzugefügt und nicht effektiv in der Wissensdatenbank gespeichert

Wie kann man eine neue Regel an erster und letzter Stelle zur Laufzeit hinzufügen?

- `asserta/1` : Regel als erste Regel hinzufügen
- `assertz/1` : Regel als letzte Regel hinzufügen
- `asserta(<Regel hier einfügen>)`

Wie kann ich eine Regel zur Laufzeit entfernen?

- Mit dem Prädikat `retract/1`
- z.B. `retract(bigger(me, you)).`

Was ist der Vorteil von Assertions?

- Gewisse Teilprobleme müssen nicht mehrmals gelöst werden,
- sondern die Lösungen werden mit Hilfe von Assertions in der Wissensdatenbank abgelegt

> Siehe Fibonacci-Beispiel auf Slides (19)

Definiere die Datenstruktur Liste

- endliche Sequenz von Elementen
- in Prolog mit Hilfe von [ ] (eckigen Klammern) dargestellt
- Elemente einer Liste werden in eckigen Klammern eingeschlossen und durch Komma getrennt
- Länge einer Liste: Anzahl Elemente, welche in dieser Liste enthalten sind
- Listen-Elemente: beliebige Prolog Terme (Atome, Zahlen oder Listen)
- Leere Liste: [ ]

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

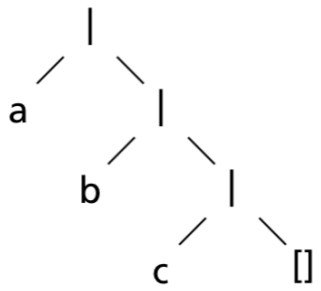
Wie sind Listen aufgebaut?

- rekursiv
- Nicht-leere Liste besteht aus
    - **Kopf** (head) = erstes Element

- **Schwanz** (tail) = Liste mit restlichen Elementen

Wie sieht die Baumdarstellung einer Liste aus?

- der Liste `[a, b, c]`
- Terminierung durch [ ] $\rightarrow$ wie Rekursionsbasis



Wie kann eine Liste in Kopf und Schwanz unterteilt werden?

- Mit dem Listen-Operator `|`

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

Wie geben wir das 3. und 4. Element einer Liste aus?

- Wir verwenden den Listen-Operator `|`
- Wir verwenden anonyme Variablen `_`, für die Element, die uns nicht interessieren

```
?- [_, _, X3, X4 | _] = [a, b, c, d, e, f, g].
X3 = c,
X4 = d.
```

Wie definiert man ein Prädikat um zu Ermitteln, ob ein Element in einer Liste vorkommt oder nicht?

- `mem/2`

- Relationen definieren
  - `mem(b, [a, b, c])` soll stimmen
  - `mem(b, [a, [b, c]])` soll nicht stimmen
  - `mem(d, [a, b, c])` soll nicht stimmen
  - `mem([b, c], [a, [b, c]])` soll stimmen
- Element `X` kommt in Liste `L` vor wenn entweder,
  - `X` der Kopf ist oder,
  - `X` im Schwanz vorkommt

```
mem(X, [X | _]).        % X ist der Kopf (Schwanz irrelevant)
mem(X, [_| Tail]) :-
    mem(X, Tail).       % X ist im Schwanz
```

> - anonyme Variablen um Prolog-Warnung **Singleton Variables** zu vermeiden
> - Prolog hat bereits ein eingebautes Prädikat `member/2`

Wie erzeugt man Permutationen mittels `mem/2` ?

- Wir beschränken uns auf Länge der Liste, um zu vermeiden, dass Prolog durch Backtracking und Teifensuche nicht (unendlich) lange Lösungslisten generiert (siehe Beispiel auf Slide 36 / 43)

```
L = [_, _], mem(a, L), mem(b, L).
L = [a, b] ;
L = [b, a] ;
false.          % Erzeugt alle 2! = 2 Permutationen
```

Wie definiert man ein Prädikat um zu Ermitteln, ob zwei Listen zusammengehängt Liste 3 ergeben?

- `conc/2`
- Relationen definieren
  - `conc([a, b], [c, d], [a, b, c, d])` soll stimmen
  - `conc([a, b], [c], [a, b, c, d])` soll nicht stimmen
  - `conc([a, b, c], [d], [a, b, c, d])` soll stimmen
  - `conc([a, b, c], [], [c, b, a])` soll nicht stimmen
- Bedingung trifft zu wenn,

- Wenn die erste Liste die leere Liste ist, dann müssen die zweite und dritte dieselbe Liste sein
- Wenn die erste Liste nicht leer ist, dann hat sie einen Kopf und einen Schwanz (`[X | L1]`). Das Resultat ist die Liste `[X | L3]` , wobei L3 die Konkatenation der Listen L1 und L2 ist

```
conc([], L, L). % Falls erste Liste leer und zweite Liste = dritte Liste
conc([X | L1], L2, [X | L3]) :-
    conc(L1, L2, L3). % Erste Liste nicht leer mit Kopf [X | L1]
```