Einführung in das Programmieren – Prolog
Sommersemester 2006

Teil 4: Rekursion und Listen

Version 1.0

Gliederung der LV

Teil 1: Ein motivierendes Beispiel

Teil 2: Einführung und Grundkonzepte
- Syntax, Regeln, Unifikation, Abarbeitung

Teil 3: Arithmetik

Teil 4: Rekursion und Listen

Teil 5: Programmfluß
- Negation, Cut

Teil 6: Verschiedenes
- Ein-/Ausgabe, Programmierstil

Teil 7: Wissensbasis
- Löschen und Hinzufügen von Klauseln

Teil 8: Fortgeschrittene Techniken
- Metainterpreter, iterative Deepening, PTTP, Differenzlisten, doppelt verkettete Listen

Rekursion

- Rekursion ist wichtiges Konzept bei der Entwicklung von Programmen in Prolog
- Rekursion auch wichtiges Konzept generell in der Informatik und auch Mathematik
- Ähnlich der vollständigen Induktion in der Mathematik

Prinzip der Rekursion

- Um ein komplexes Problem zu lösen, gib an
  1. die Lösung für das einfachste Problem dieser Art und
  2. eine Regel, um ein komplexes Problem in ein einfacheres Problem zu transformieren

Beispiel: Berechnung von $n!$

- rekursive Definition von $n!$

$$1! = 1$$
$$n! = (n-1)! \cdot n \quad \text{für } n > 1$$

- In Prolog

```
fakultaet(1,1).
fakultaet(N, Result) :-
            N > 1,
            N1 is N-1,
            fakultaet(N1, Result1),
            Result is N * Result1.
```

Listen

- eine Liste ist eine Aneinanderreihung beliebig vieler Elemente
- eine Liste ist eingeschlossen in eckigen Klammern
  Beispiele:
  ```
  [elefant, pferd, esel, hund]
  [a, X, [], f(X,y), 89, [w,e,r], aunt(X,hilde)]
  ```
- Elemente der Liste sind Terme
- Reihenfolge der Listenelemente ist wesentlich
- eine Liste kann selbst wieder Listen enthalten
- eine Liste ist ein Term

Interne Repräsentation von Listen:

- Listen sind Terme mit dem Funktor . (Punkt) und dem speziellen Atom [ ] als ein Argument auf innerstem Niveau.
  Die eckigen Klammern sind ”nur” eine schöne Notation.
- *Intern* entspricht die Liste
  ```
  [a, b, c]
  ```
  dem Term
  ```
  .(a, .(b, .(c, [])))
  ```

Head-Tail Notation

- Der | zerlegt eine nicht leere Liste in den *Kopf* und die *Restliste*.

```
[Kopf|Rest]
```

- Vor dem | können auch mehrere Elemente stehen

```
[a | [b, c]] = [a, b, c]
[a, b | [c]] = [a, b, c]
[a, b, c | []] = [a, b, c]
```

Einige Beispiele

Stimmen die Gleichheiten?

```
[b, a, d] = [d, a, b].
[a, b, d] = [X].
[a, [b, d]] = [a, b, d].
[a, b, c, d] = [A | L].
[a, b, c, d] = [A, L].
[a, b, c, d] = [A, B | L].
[[a, b], c, d] = [A | L].
[a, b] = [A | L].
[a, b] = [A, L].
[a] = [A, L].
[] = [A, L].
```

Listen und Rekursion

- Listen sind rekursive Datenstrukturen:
  sind leer oder bestehen aus Kopf und Rest, der wieder Liste ist.

$\rightsquigarrow$ Operationen über Listen werden auch rekursiv sein.

- Oft zwei Klauseln zu definieren; eine deckt den rekursiven Fall, die andere den Basisfall ab

Beispiele

element(E,L)

- Teste of ein Element E schon in einer Liste L vorhanden ist.

```
element(E, [E | Rest]).
element(E, [Kopf | Rest]) :- element(E, Rest).
```

Verwendungsmöglichkeiten

1. als Zugehörigkeitstest
2. zur Erzeugung aller Elemente einer Liste
3. Erzeugung von Listen, die bestimmtes Element enthalten

Gibt es auch als Built-in Prädikat: member

Verketten von Listen

append

- `append(L1, L2, L3)` gelingt, falls die Verkettung der Listen `L1` und `L2` dir Liste `L3` ergibt.
- beispielhaftes Verhalten

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

- beispielhaftes Verhalten

```
append([1,2], [3], [1,2,3])).
Yes

append([1,2], [3,4], L)).
L = [1,2,3,4]

append([1,2], L, [1,2,3,4,5])).
L = [3,4,5]

append(L, [4,5], [1,2,3,4,5])).
L = [1,2,3]
```

Beispiel

- Ausgabe aller Elemente einer Liste

```
show (Liste) :-
        element(Element, Liste),
        write(Element),
        nl,
        fail.
```

- Beachte: `fail` ist ein Standardprädikat, das immer fehlschlägt.
- Was geschieht bei folgender Eingabe?
  `[elefant, pferd, esel, hund]`

Beispiel

```
?- show([elefant, pferd, esel, hund]).
elefant
pferd
esel
hund
No.
```

- Das `fail` veranlaßt Backtracking
- Teilziel `element(Element, Liste)` ist als einziges *backtrackfähig*
- Bei jedem Backtrack-Schritt wird neues Element der Liste mit Variable `Element` unifiziert.
- Wenn Liste vollständig abgearbeitet ist: Fehlschlag, `No`

Beispiele

- `nextto(X,Y,L)` ist erfüllt, falls `X` und `Y` in Liste `L` nebeneinander liegen.

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

- Prädikat `ungerade(L)` überprüft, ob Anzahl Elemente in Liste L ungerade ist

```
ungerade([_]).
ungerade(_, _ | L) :- ungerade(L).
```

Built-In-Prädikate für Listen

is_list, **proper_list**

last

length

append

member, **select**, **nth**

sublist

reverse

sort

Effizienzbetrachtungen

- Listen sind rekursive Datenstrukturen
- ihr allgemeiner Charakter und damit verbundene universelle Anwendbarkeit führt leicht zu uneffektiven Programmen.
- Wissen über interne Darstellung von Listen kann helfen, Ineffektivitäten zu vermeiden
- Zwei wichtige Techniken: Akkumulatoren und Differenzlisten

Akkumulatoren

- Oftmals sind Datenstrukturen zu durchforsten und ein bestimmtes Ergebis zu erzielen, berechnen, ..
- *Akkumulatoren* sind zusätzliche Argumente eines Prädikats, um Zwischenergebnisse zu speichern

Beispiel

- Berechnung der Länge einer Liste

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

- Beim rekursiven Aufruf kein Zwischenergebnis bekannt
- Andere Möglichkeit: akkumuliere Antwort in jedem rekursiven Aufruf

```
laenge(L,N) :- laenge(L, 0, N).

laenge([ ], N, N).
laenge([_|T], A, N) :-
            A1 is A + 1,
            laenge(T,A1,N).
```

Sinnvolleres Beispiel: reverse

- Realisiere ein Prädikat `reverse(Liste,UmgedrehteListe)`, das die Reihenfolge der Elemente einer Liste umdreht.
- Lösungsmöglichkeit
  1. Kehrt man die leere Liste um, erhält man wieder die leere Liste
  2. eine nichtleere Liste `[X|L1]` kehrt man um, indem man `L1` umkehrt und daran einelementige Liste `[X]` kettet.

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

Problem: `append` muß jedes Mal die neue Liste durchlaufen und kopieren, um ein neues Element anzuhängen.

Sinnvolleres Beispiel: reverse

- bessere Lösung: mit Akkumulator:

```
reverse(Lin, Lout):-
            reverse(Lin, [ ], Lout).

reverse([ ], Bisher, Bisher).
reverse([X | L1], Bisher, L2) :-
            reverse(L1, [X | Bisher], L2).
```

- Zusätzlicher Vorteil: ***tail recursion***
  - der Rekursionsabstieg findet im letzten Prädikat des Rumpfes statt
  - der Compiler kann nun unter bestimmten Bedingungen den Stackframe der aktuellen Prozedur durch den neuen überschreiben

  $\longrightarrow$ Speicher- und Geschwindigkeitsgewinn