Übungen 9: Rekursive Programmiertechniken

Programmiertechniken in der Computerlinguistik I · Wintersemester 2004/2005

Um keine Zeit mit Tippen und Tippfehlerkorrektur zu verschwenden, findest du Programmtexte dieser Übungen unter http://www.cl.unizh.ch /siclemat/lehre/ws0405/pcl1/uebung9.txt

1. Hyponymie linksrekursiv

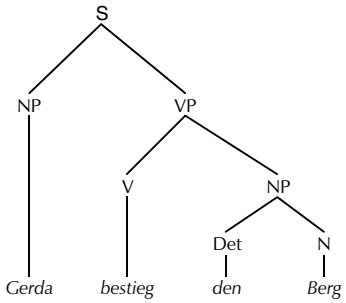
Kann die Relation hypo/2 aus der Vorlesung so geschrieben werden, dass Linksrekursion erscheint, die den Prolog-Interpreter in unendlichen Suchästen verschwinden lassen?

a) Falls ja, wie müsste die Definition von hypo/2 aussehen?

b) Falls ja, für welche Typen von Anfragen meldet sich der Prolog-Interpreter nicht mehr mit einer Antwort zurück? (Ausprobieren!)

2. Dominanz in Satzstrukturen

In der Linguistik werden Satzstrukturen oft in Form von hierarchischen Bäumen repräsentiert. Man spricht insbesondere von der Dominanz-Relation, welche die einzelnen Knoten des Baums ordnet. Als Beispiel diene hier die Struktur des Satzes "Gerda bestieg den Berg".



a) Repräsentiere die unmittelbaren Dominanzverhältnisse des Beispielsatzes in Form von Fakten wie

```
dominiert_unmittelbar(s, np).
dominiert_unmittelbar(det, den).
```

Was soll mit Knoten gemacht werden, die mehrmals im Baum vorkommen?

b) Definiere ein Prädikat dominiert/2, das gelingt, falls das erste Argument das zweite Argument dominiert.

```
?- dominiert(s, den).
yes
```

c) Schreibe ein Prädikat wurzel/1, das nur für diejenigen Knoten wahr ist, die durch keinen Knoten dominiert werden. Kann dein Prädikat auch im Modus wurzel(–Atom) verwendet werden?

3. Parsen mit append/3

a) Erweitere die Grammatik aus den Folien um die Regel

NP → Det N

mit den entsprechenden Lexikoneinträgen, damit "der Hund" analysiert werden kann. Überprüfe deinen Parser mit Anfragen für die (in-)korrekten Sätze "Der Hund frisst" und "Hund der frisst".
b) Versuche exakt zu verstehen, wie der Prolog-Interpreter auf seine Lösung kommt. Zeichne einen Beweisbaum oder mache ein Tracing für obige Anfragen, bzw. ein Debugging auf append/3 (vgl. Folien des Kapitels "Debuggen"), wenn es zu unübersichtlich wird. Wie beurteilst du die Effizienz dieses Verfahrens?

4. Naiv und weniger naiv

Experiment: Teste auf deinem Computer aus, bei welcher Anzahl Elemente das Umkehren einer Liste auf die naive Art etwa 3 Sekunden dauert. Wie ist die Laufzeit der Listenumkehrung mit Akkumulator? Teste danach mit 10-facher Anzahl.
Hinweis: Mit dem vordefinierten length/2 kann eine Liste beliebiger Grösse konstruiert werden. Beispielanfrage:

```
?- length(Liste, 759), reverse(Liste, Umkehrung).
```

5. Wort nach Komma

Definiere das rekursive Prädikat “Wort nach Komma“ – kurz wnk/2, das genau dann wahr ist, wenn das 1. Argument eine Wort-/Interpunktionsliste und das 2. Argument ein Wort ist, das im Satz nach einem Komma steht.

Das Prädikat soll durch Backtracking alle möglichen Lösungen liefern und wird mit instantiiertem 1. Argument aufgerufen.

```
?- wnk(['Ja',',',du,hast,recht,mit,dem,',',was,du,denkst,'.'], WNK).
WNK = du? ;
WNK = was? ;
no
```

6. Palindrome

Wörter (oder Sätze), die gleich lauten, auch wenn sie umgekehrt gelesen werden, heissen *Palindrome*. Schreibe ein Prolog-Prädikat palindrom/1, das feststellt, ob die Umkehrung einer Liste gleich wie die ursprüngliche Liste ist.
Beispiel für die Benutzung des Prädikates:

```
?- palindrom([k,a,j,a,k]).
yes
?- palindrom([k,a,n,u]).
no
```

Tipp: Zuerst studieren, dann ganz wenig programmieren!