Lösungen: Einführung

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

2. Schnurrli Meier

Nur die fett gedruckten Zeilen sind neu.

```
/* Personen */
person(hans).
person(klara).
person(sabrina).
person(kevin).

/* Tiere */
hund(fido).
katze(schnurrli).

/* Geschlecht */
weiblich(klara).
weiblich(sabrina).
weiblich(schnurrli).
maennlich(hans).
maennlich(kevin).
maennlich(fido).

/* Wenn jemand eine Person und weiblich ist, handelt es sich
   um eine Frau. */
frau(Jemand) :-
  person(Jemand),
  weiblich(Jemand).
```

3. Regeln umschreiben

Formuliere folgende Prolog-Regeln als deutsche Sätze:

a) tier(Etwas) :- hund(Etwas). — Wenn etwas ein Hund ist, ist es auch ein Tier.
b) tier(Etwas) :- katze(Etwas). — Alle Katzen sind Tiere.
c) bestraft(leben, X) :- kommt_zu_spaet(X). — Wer zu spät kommt, den bestraft das Leben.
d) faellt_in(X, Z) :- graebt(X, Y, Z), ungleich(X, Y), grube(Z). — Wer anderen eine Grube fällt, der fällt selbst hinein.

4. Formalisieren

Schreibe die folgenden Aussagen bzw. Regeln als Prolog-Fakten:

a) Peter kennt Katrin.
   `kennt(peter, katrin).`
   Alternativ: `kennt(katrin, peter).`

b) Katrin kennt Peter.
`kennt(katrin, peter).`
Alternativ: `kennt(peter, katrin).`

c) Fido mag Gulasch.
`mag(fido, gulasch).`
Alternativ: `mag(gulasch, fido).`

Prolog gibt den einzelnen Argumentstellen keine besondere Bedeutung. Nichts schreibt Dir also vor, z.B. das Subjekt an die erste Stelle zu setzen. Wichtig ist aber, dass Du konsistent bist, also nicht etwa mag(fido, gulasch) zusammen mit mag(fisch, schnurrli) verwendest.

Auch die benutzten Namen sind prinzipiell egal — Du könntest zum Beispiel auch »kennen« statt »kennt« schreiben.

d) Olten liegt zwischen Zürich und Genf.
`zwischen(zuerich, olten, genf).`
Dies ist natürlich nicht die einzig mögliche Lösung! Siehe die Bemerkungen oben.

e) Eine Person, die männlich ist, ist ein Mann.
```
mann(X) :-
  person(X),
  maennlich(X).
```

Wie lauten die folgenden Anfragen in Prolog?

g) Kennt Peter Katrin?
`kennt(peter, katrin).`
Alternativ: `kennt(katrin, peter).`
Du musst für Subjekt und Prädikat dieselben Argumentstellen wie oben verwenden, also eben konsistent sein.

h) Wen kennt Peter?
`kennt(peter, Wen).`
Alternativ: `kennt(Wen, peter).`

i) Was mag Fido?
`mag(fido, Was).`
Alternativ: `mag(Was, fido).`

5. Fakten und Regeln

Handelt es sich bei den folgenden Klauseln jeweils um Fakten oder um Regeln?

a) idiot(peter). — Fakt
b) genie(maria). — Fakt
c) fachidiot(Jemand) :- idiot(Jemand), genie(Jemand). — Regel
d) tochter(T, Elternteil) :- mutter(Elternteil, T), weiblich(T). — Regel
e) tochter(T, V) :- vater(V, T), weiblich(T). — Regel

Lösungen: Strukturen/Ablauf

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Terme klassifizieren

a) `hans` — Atom
b) `Klara` — Variable
c) `'Kevin'` — Atom (zwischen Apostrophen)
d) `s a b r i n a` — unzulässig
e) `' s a b r i n a '` — Atom (zwischen Apostrophen)
f) `s_a_b_r_i_n_a_` — Atom (beginnt mit Kleinbuchstabe)
g) `_s_a_b_r_i_n_a` — Variable (beginnt mit Unterstrich)
h) `S_a_b_r_i_n_a_` — Variable (beg. mit Grossbuchtabe)
i) `_S_a_b_r_i_n_a_` — Variable (beginnt mit Unterstrich)
j) `liebt(hans, klara)`
Komplexer Term liebt/2, 1. Argument: Atom, 2. Arg.: Atom
k) `liebt(hans, Klara)`
Komplexer Term liebt/2, 1. Arg.: Atom, 2. Arg.: Variable
l) `bUCH` — Atom (beginnt mit Kleinbuchtabe)
m) `Buch` — Variable (beginnt mit Grossbuchst.)
n) `familie(hans, klara, sabrina, kevin)`
Komplexer Term familie/4, alle Argumente Atome
o) `beisst(schnurrli fido)` — unzulässig
p) `kratzt(Schnurrli, Fido)`
Komplexer Term kratzt/2, alle Argumente Variablen
q) `gelogen(beisst(schnurrli, FIDO))`
Komplexer Term gelogen/1, einziges Argument ist komplexer Term beisst/2, dessen erstes Argument ein Atom und dessen zweites Argument eine Variable ist
r) `Gelogen(beisst(schnurrli, FIDO))` — unzulässig
s) `behauptung(hans, gelogen(beisst(schnurrli, 'Fido')))`
Komplexer Term behauptung/2, erstes Argument Atom,
zweites Argument: komplexer Term gelogen/1, dessen einziges Argument komplexer Term beisst/2, dessen beide Argumente Atome
t) `muehsam(muessen(viele(terme), klassifizieren))`
Komplexer Term muehsam/1, einziges Argument komplexer Term muessen/2, dessen erstes Argument komplexer Term viele/1, dessen einziges Argument Atom. Zweites Argument von »muessen« ist Atom.

2. Unifizieren

a)	`brot = brot`	ja
b)	`'Brot' = brot`	nein
c)	`'brot' = brot`	ja
d)	`Brot = brot`	ja, Brot / brot
e)	`brot = wurst`	nein
f)	`essen(brot) = wurst`	nein
g)	`essen(brot) = X`	ja, X / essen(brot)
h)	`essen(X) = essen(brot)`	ja, X / brot
i)	`essen(brot, X) = essen(Y, wurst)`	ja, Y / brot und X / wurst
j)	`essen(brot, X, wasser) = essen(brot, Z, Z)`	ja, X / Z und Z / wasser, somit auch X / wasser
k)	`essen(brot, X, wasser) = essen(Y, Z, Z)`	ja, Y / brot und X / Z und Z / wasser
l)	`essen(brot, X, wasser) = essen(Z, Z, Z)`	nein
m)	`essen(X) = essen(brot, wasser)`	nein
n)	`mahlzeit(essen(brot), trinken(wasser)) = mahlzeit(X, Y)`	ja, X / essen(brot) und Y / trinken(wasser)
o)	`X = X`	ja
p)	`X = Y`	ja, X / Y [oder: Y / X]

3. Gesund ist, wer Früchte isst

Adam mag Äpfel. Friederike mag Frikadellen. Ottilie mag Orangen. Äpfel sind Früchte. Orangen sind Früchte. Wenn jemand etwas mag, und dieses Etwas ist eine Frucht, dann ist dieser Jemand gesund.

a)

```
/* mag/2 */
mag(adam, aepfel).              % Adam mag Aepfel.
mag(friederike, frikadellen).   % Friederike mag Frikadellen.
mag(ottilie, orangen).          % Ottilie mag Orangen.

/* Was ist alles eine Frucht? */
fruchtig(aepfel).
fruchtig(orangen).

/* Gesund ist jemand, wenn er/sie etwas mag, das fruchtig ist. */
gesund(Jemand) :-
  mag(Jemand, HamHam),
  fruchtig(HamHam).
```

b) `?- gesund(Wer).`

c) Durch Eingabe eines Strichpunkts, wodurch Backtracking erzwungen wird.

Lösungen: Rekursion

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

3. Patriarchat

```
/* a) Die Vater-Sohn-Beziehung bei Familie Meier:
      vater(Vater, Sohn) */

vater(hans, kevin).
vater(willibald, hans).
vater(herrmann, willibald).
vater(gottfried, herrmann).
vater(albrecht, gottfried).
```

```
/* b) Nicht-rekursive Definition von grossvater/2 */

grossvater(Opa, Spross) :-
  vater(Opa, Papi),
  vater(Papi, Spross).
```

```
/* b) Nicht-rekursive Definition von urgrossvater/2 */

urgrossvater(Uropa, Spross) :-
  vater(Uropa, Opa),
  vater(Opa, Papi),
  vater(Papi, Spross).
```

```
/* b) Nicht-rekursive Definition von ururgrossvater/2 */

ururgrossvater(Ururopa, Spross) :-
  vater(Ururopa, Uropa),
  vater(Uropa, Opa),
  vater(Opa, Papi),
  vater(Papi, Spross).
```

```
/* c) Alternative Definitionen von urgrossvater/2 etc. */

urgrossvater(Uropa, Spross) :-
  vater(Papi, Spross),
  grossvater(Uropa, Papi).

ururgrossvater(Ururopa, Spross) :-
  vater(Papi, Spross),
  urgrossvater(Ururopa, Papi).
```

```
/* d) Rekursive Definition von vorfahr/2 */

vorfahr(Papi, Spross) :-   % Abbruchbedingung
  vater(Papi, Spross).

vorfahr(Ahn, Spross) :-    % rekursiver Fall
  vater(Papi, Spross),
  vorfahr(Ahn, Papi).
```

4. Tiere im Bahnhof

Bei den Tieren in rekursiver Umgebung handelt es sich um die umstrittene Installation *Das Philosophische Ei* von Mario Merz. Plastiktiere (besonders hervorstechend: der Hirsch) sind vor einer rot leuchtenden Spirale aufgehängt. Weiterer Bestandteil ist eine Zahlenreihe aus blauen Leuchtstoffröhren:

1 – 1 – 2 – 3 – 5 – 8 – 13 – 21 – 34 – 55

Diese Zahlen sind der Anfang der sogenannten Fibonacci-Zahlen, die folgendermassen gebildet werden:

Abbruchbedingung:	$\text{fib}(0) = \text{fib}(1) = 1$
Rekursiver Fall (für $n \geq 2$):	$\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$

Lösungen: Listen

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Listen unifizieren

Gib — ohne den Computer zu benutzen — an, was das Ergebnis der folgenden Anfragen sein wird. Überprüfe anschliessend am Computer, ob Prolog tatsächlich die erwartete Antwort gibt.

a) `?- X = [fido, gulasch].`
`X = [fido, gulasch]` — *Da eine Liste ein gewöhnlicher Term ist, kann sie auch mit einer Variablen unifiziert werden.*

b) `?- X = [fido | [gulasch]].`
`X = [fido, gulasch]` — *Die Liste, die fido als Anfang und [gulasch] als Rest besitzt (vgl. das Beispiel mit [kaffee, tee, cola] auf den Folienkopien).*

c) `?- [fido, frisst, Was] = [Wer, frisst, gulasch].`
`Was = gulasch, Wer = fido`

d) `?- [fido | Rest] = [fido, frisst, gulasch].`
`Rest = [frisst, gulasch]` — *Eben der Rest von [fido, frisst, gulasch]*

e) `?- [fido, Rest] = [fido, frisst, gulasch].`
no — *Die Unifikation schlägt fehl, weil links zwei und rechts drei Elemente stehen. Bei Aufgabe d) dagegen wurde auf der linken Seite nichts über die Anzahl der Elemente in der Liste ausgesagt, abgesehen davon, dass sie mindestens ein Element besitzen muss, das mit dem Atom fido unifizierbar ist.*

f) `?- [fido, frisst | [X]] = [fido, frisst, gulasch | []].`
`X = gulasch` — *[fido, frisst | [X]] ist dasselbe wie [fido, frisst, X], weil eine ein-elementige Rest-Liste dasselbe wie ein weiteres Element ist. Die Angabe der leeren Liste als Rest auf der rechten Seite ist überflüssig, weil jede gut erzogene Liste mit der leeren Liste aufhört.*

3. Element aus einer Liste ersetzen

Schreibe ein nicht-rekursives Prolog-Prädikat, welches das dritte Element einer Liste mit mindestens drei Elementen durch einen anderen Term ersetzt.

```
% ersetze3/3: Ersetzt das dritte Element einer mindestens
% drei-elementigen Liste durch einen anderen Term.
% Benutzung: ersetze3(AlteListe, TermFuer3, Ergebnis)

ersetze3([Erstes, Zweites, _   | Rest],
         Neu,
         [Erstes, Zweites, Neu | Rest]).
```

4. Fido mag kein Gulasch mehr

Schreibe ein Prolog-Prädikat, das rekursiv eine Liste durchläuft und alle Vorkommen eines Terms durch einen anderen ersetzt.

```
/************************************************************/
/* ersetze(TermAlt, TermNeu, AlteListe, NeueListe)          */
/*                                                          */
/* Ersetzt alle Vorkommen des Terms TermAlt in AlteListe    */
/* durch TermNeu. Das Ergebnis wird in NeueListe zurueck-   */
/* gegeben. Das Praedikat funktioniert auch “umgekehrt”:    */
/* ?- ersetze(X, Y, [fido,f,gulasch], [fido,f,sellerie]).  */
/* X = gulasch, Y = sellerie                                */
/************************************************************/

% Abbruchbedingung: Die leere Liste wird auf die leere Liste
% abgebildet. Dabei spielen TermAlt und TermNeu keine Rolle,
% weswegen an den entsprechenden Argumentstellen die anonyme
% Variable (_) steht.

ersetze(_, _, [], []).


% Rekursiver Fall 1: Das erste Element der Liste ist mit
% TermAlt unifizierbar. Bilde den Rest der Liste rekursiv ab
% (AltRest wird auf NeuRest abgebildet, im Rumpf der Regel).
% Das Ergebnis ist dann eine Liste, die aus TermNeu als erstem
% Element und NeuRest als Rest besteht.

ersetze(TermAlt, TermNeu, [TermAlt | AltRest],
                          [TermNeu | NeuRest]) :-
   ersetze(TermAlt, TermNeu, AltRest, NeuRest).


% Rekursiver Fall 2: Wenn das erste Element der Liste mit
% TermAlt unifizierbar war, kam die zweite Klausel von ersetze/4
% zum Zug (d.h. Rekursiver Fall 1), weil Prolog immer die erste
% passende Klausel nimmt. Diese Klausel hier deckt also jene
% Faelle ab, wo das erste Element der Liste nicht mit TermAlt
% unifiziert.
% Bilde in diesem Fall wie oben den Rest der Liste rekursiv ab
% (AltRest wird auf NeuRest abgebildet, im Rumpf der Regel).
% Das Ergebnis besitzt als erstes Element das erste Element
% der Eingabe (d.h. Term) und als Rest NeuRest.

ersetze(TermAlt, TermNeu, [Term | AltRest], [Term | NeuRest]) :-
   ersetze(TermAlt, TermNeu, AltRest, NeuRest).
```

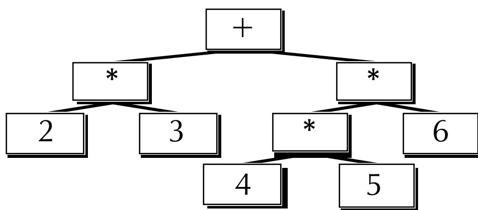
Lösungen: Arithmetik

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Baum malen

Zeichne, analog zu den Beispielen in der Vorlesung, einen Baum für die Struktur des nachfolgenden Terms.

```
2 * 3 + 4 * 5 * 6
```



2. Letztes Element einer Liste

Schreibe ein rekursives Prädikat letztes/2, welches das letzte Element einer Liste bestimmt.

```
% Abbruchbedingung: Bei einer Liste mit nur einem Element
% ist eben dieses einzige Element das Ergebnis.
letztes([Ergebnis], Ergebnis).

% Rekursiver Fall: Das letzte Element einer Liste ist das
% letzte Element im Rest der Liste.
letztes([_ | Rest], Ergebnis) :-
  letztes(Rest, Ergebnis).
```

3. Fibonacci-Zahlen

Schreibe ein rekursives Prolog-Prädikat fib/2, das für ein $n \geq 0$ die entsprechende Fibonacci-Zahl berechnet.

```
% ----------------------------------------------------------
%  fib(+N, ?Ergebnis)
% ----------------------------------------------------------
% Berechnet die Fibonacci-Zahl zu einem gegebenen N.
% Das erste Argument muss eine Zahl sein (und nicht eine
% freie Variable -- dies wird oft mit "+" geschrieben);
% das zweite Argument wird, wenn es eine freie Variable
% ist, an das Ergebnis gebunden. Wenn das zweite Argument
% eine Zahl ist, gelingt die Anfrage, wenn die Zahl gleich
% dem Ergebnis ist; sonst schlaegt sie fehl.

fib(0, 1).            % Abbruchbedingung: fib(0) = 1

fib(1, 1).            % Noch eine Abbruchbedingung: fib(1) = 1

fib(N, Ergebnis) :-   % rekursiver Fall fuer N >= 2
   N >= 2,

   N_minus_2 is N - 2,             % berechne N - 2
   fib(N_minus_2, Fib_N_minus_2), % ... und rekursiv fib(N - 2)

   N_minus_1 is N - 1,             % berechne N - 1
   fib(N_minus_1, Fib_N_minus_1), % ... und rekursiv fib(N - 1)

   Ergebnis is Fib_N_minus_2 + Fib_N_minus_1.
```

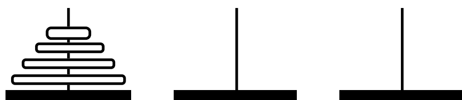
Zugegebenermassen ist dieses Programm ziemlich umständlich, und in anderen Programmiersprachen wäre das Berechnen der Fibonacci-Zahlen wesentlich eleganter ausdrückbar. Allerdings war Prolog auch nie als Werkzeug für Zahlenbeigerei (»Number Crunching«) gedacht.

Lösungen: Repetition

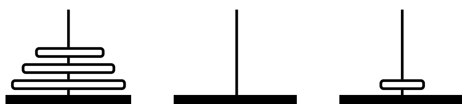
Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Die Türme von Hanoi

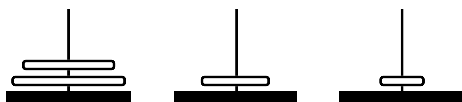
a) Spiele das Spiel mit vier Scheiben durch.



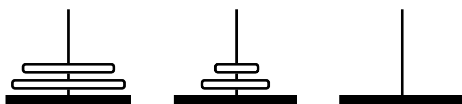
Ausgangslage



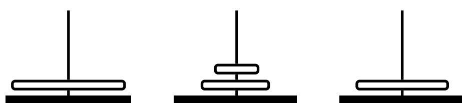
1. *links* nach *rechts*



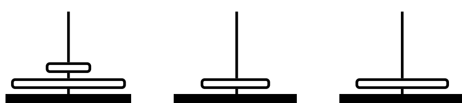
2. *links* nach *mitte*



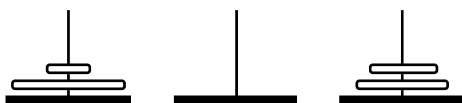
3. *rechts* nach *mitte*



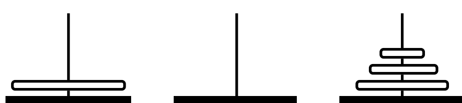
4. *links* nach *rechts*



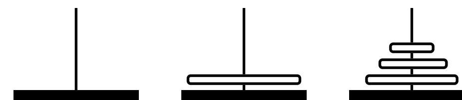
5. *mitte* nach *links*



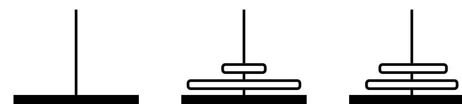
6. *mitte* nach *rechts*



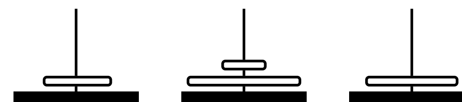
7. *links* nach *rechts*



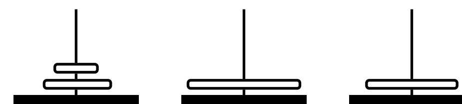
8. *links* nach *mitte*



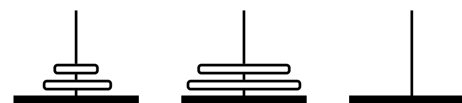
9. *rechts* nach *mitte*



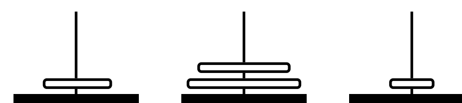
10. *rechts* nach *links*



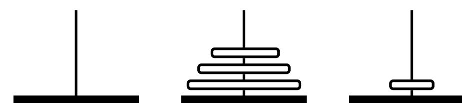
11. *mitte* nach *links*



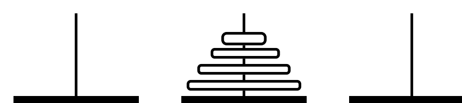
12. *rechts* nach *mitte*



13. *links* nach *rechts*



14. *links* nach *mitte*



15. *rechts* nach *mitte*

2. Prinzen von Wales

```
regiert(rhodri, 844, 878).
regiert(anarawd, 878, 916).
regiert(hywel_dda, 916, 950).
regiert(lago_ap_idwal, 950, 979).
regiert(hywel_ap_ieuaf, 979, 985).
regiert(cadwallon, 985, 986).
regiert(maredudd, 986, 999).

prinz(P, J) :-
   regiert(P, Von, Bis),
   J >= Von,
   J =< Bis.
```

3. Einer-Listen

```
einer_liste(X, [X]).
```

4. Palindrome

```
palindrom(Liste) :-      % Eine Liste ist dann ein Palindrom, …
  reverse(Liste, Liste). % … wenn sie umgekehrt gleich lautet.
```

5. Teilmengen

Schreibe ein Prolog-Prädikat `teilmenge/2`, das feststellt, ob eine Menge Teilmenge einer anderen ist.

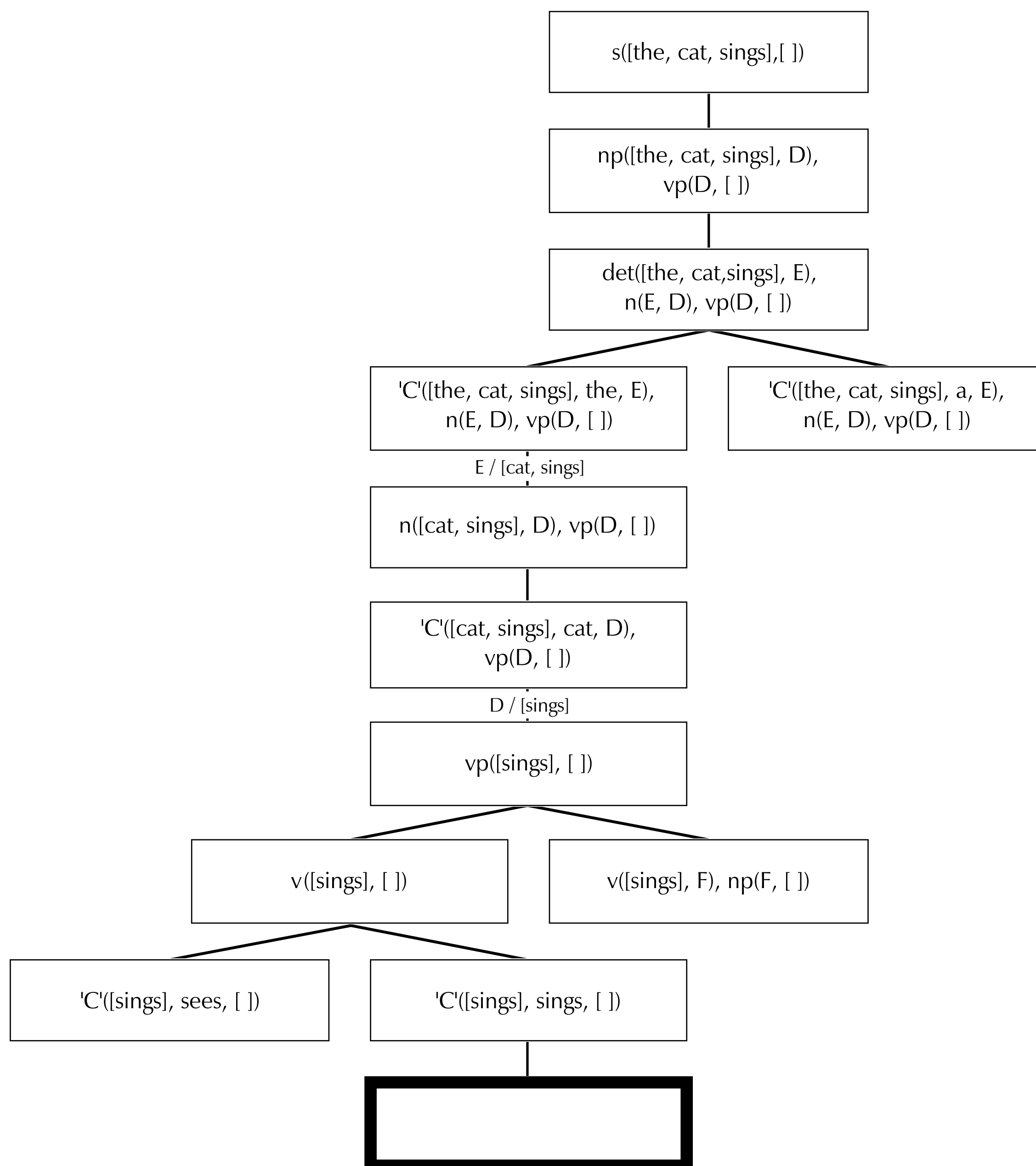
```
/* Die leere Menge ist Teil jeder Menge. */
teilmenge([], _).

/* Eine Menge ist Teilmenge einer Menge M, wenn jedes
   ihrer Elemente auch Element von M ist.
   Spalte das erste Element ab und prüfe mit member/2,
   ob es Element von M ist. Der Rest wird rekursiv
   geprüft.
*/
teilmenge([ErstesElement|Rest], M) :-
  member(ErstesElement, M),
  teilmenge(Rest, M).

/* member/2 -- siehe Folienkopien zum Thema "Listen". */
member(X, [X|_]).
member(X, [_|Rest]) :-
  member(X, Rest).
```

Lösungen: DCGs

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Singende Katzen

Angenommen, die Katzen-DCG wäre bereits konsultiert worden: Welche einzelnen Schritte nimmt Prolog vor, um die Anfrage `s([the,cat,sings], [])` zu beweisen?



2. Tilgungsregeln

SICStus Prolog übersetzt die DCG-Regel

```
leer --> [].
```

zu folgender Klausel:

```
leer(A, B) :- B = A.
```

Tilgungsregeln sind für Top-Down-Parser unproblematisch: Das »Bedürfnis« nach einer Konstituente, zu der eine Tilgungsregel gibt, kann einfach erfüllt werden, ohne irgendwelche Terminalsymbole zu konsumieren.

3. Reden ist Silber

Prolog gibt sämtliche Sätze aus, die gemäss der Grammatik zulässig sind. Die Menge dieser Sätze zu einer Grammatik *G* wird übrigens oft *Sprache der Grammatik* oder abgekürzt *L*(*G*) genannt.

Diese Menge kann auch unendlich gross sein, wenn die Grammatik rekursive Regeln enthält. In diesem Fall kann beliebig oft Backtracking ausgelöst werden, und es werden immer wieder neue Lösungen gefunden.

5. Linksrekursion

Ein Top-Down-Parser muss nicht unbedingt mit dem ersten Wort des Satzes anfangen und sich von links nach rechts vorarbeiten. Ebensogut könnte er mit dem letzten Wort beginnen und sich zum Satzanfang (also von rechts nach links) durcharbeiten — die prinzipielle Verarbeitungsrichtung ist dennoch top-down, d.h. der Parser ginge vom Startsymbol aus und arbeitet sich zu den Terminalsymbolen durch. Dies kann entweder durch geeignete Modifikationen am Parser bewerkstelligt werden, oder man kehrt die Reihenfolge der Symbole bei allen rechten Seiten der Grammatikregeln ebenso um wie die Eingabesätze. Allerdings funktioniert diese Technik dann nicht bei rechts-rekursiven Grammatiken.

Eine andere Möglichkeit ist, die linksrekursive Grammatik *G* in eine leichter parsbare Form *G′* umzuwandeln und zusätzlich eine Methode bereitzustellen, für jeden Strukturbaum zu *G′* einen entsprechenden zu *G* zu erzeugen. Der Parser arbeitet dann mit der nicht-linksrekursiven Grammatik *G′*, dem Benutzer werden aber Bäume zur ursprünglichen Grammatik *G* gezeigt.

Falls die Grammatik keine Tilgungsregeln enthält (d.h. Regeln mit ε auf der rechten Seite), kann eine Ableitung nicht mehr Symbole enthalten als der Eingabesatz Wörter. Dies kann benutzt werden, um die Suchtiefe zu begrenzen.

In jedem Fall kann dann nicht der in Prolog eingebaute Parser benutzt werden, sondern es muss ein eigenes Programm entwickelt werden.

Lösungen: Shift-Reduce-Parsing

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Keller-Operationen

Schreibe Prolog-Prädikate für die Operationen auf einem Keller.

a) `push(Element, Stack, [Element|Stack]).`

b) `pop([Top|Stack], Top, Stack).`
Beim Versuch, etwas von einem leeren Keller zu nehmen, schlägt die Anfrage fehl, da `[Top|Stack]` nicht mit der leeren Liste `[]` (welche für einen leeren Keller steht) unifizierbar ist.

2. Shift-Reduce-Parsing

Führe auf dem Papier ein Shift-Reduce-Parsing der folgenden Eingabeketten durch. Gib für jeden Shift- oder Reduce-Schritt an, wie Keller und Eingabekette aussehen.

a) the cat sees a dog

Schritt	Aktion	Keller	Eingabekette
0	—	ε	the cat sees a dog
1	shift	the	cat sees a dog
2	reduce	Det	cat sees a dog
3	shift	Det cat	sees a dog
4	reduce	Det N	sees a dog
5	reduce	NP	sees a dog
6	shift	NP sees	a dog
7	reduce	NP V	a dog
8	reduce	NP VP	a dog
9	reduce	S	a dog
10	shift	S a	dog
11	reduce	S Det	dog
12	shift	S Det dog	ε
13	reduce	S Det N	ε
14	reduce	S NP	ε

Zwar konnte die gesamte Eingabekette konsumiert werden, aber auf dem Keller steht nicht ein einzelnes S. Also muss Backtracking erfolgen. In Schritt 7 hätte auch die Regel *VP* → *V NP* anstelle von *VP* → *V* genommen werden können:

7	shift	NP V a	dog
8	reduce	NP V Det	dog
9	shift	NP V Det dog	ε
10	reduce	NP V Det N	ε
11	reduce	NP V NP	ε
12	reduce	NP VP	ε
13	reduce	S	ε

b) the dog the dog

Schritt	Aktion	Keller	Eingabekette
0	—	ε	the dog the dog
1	shift	the	dog the dog
2	reduce	Det	dog the dog
3	shift	Det dog	the dog
4	reduce	Det N	the dog
5	reduce	NP	the dog
6	shift	NP the	dog
7	reduce	NP Det	dog
8	shift	NP Det dog	ε
9	reduce	NP Det N	ε
10	reduce	NP NP	ε

3. Strukturbäume

Welche Änderungen sind nötig, damit die Struktur in der selben Form wie früher bei den DCGs ausgegeben wird?

```
% -------------------------------------------------------------
% Phrase Structure Rules
% -------------------------------------------------------------

% S --> NP VP
brule([vp(VPStruktur), np(NPStruktur) | X],
    [s(s(NPStruktur, VPStruktur)) | X]).

% NP --> Det N
brule([n(NStruktur), det(DetStruktur) | X],
    [np(np(DetStruktur, NStruktur)) | X]).

% VP --> V NP
brule([np(NPStruktur), v(VStruktur) | X],
    [vp(vp(VStruktur, NPStruktur)) | X]).

% VP --> V
brule([v(VStruktur) | X],
    [vp(vp(VStruktur)) | X]).
```

```
brule([Word   | X], [Cat | X]) :-
   word(Cat, Word).


% ------------------------------------------------------------
% Lexicon
% ------------------------------------------------------------

word(det(det(the)), the).
word(n(n(dog)), dog).
word(n(n(cat)), cat).
word(v(v(chase)), chase).
word(v(v(chases)), chases).
word(v(v(see)), see).
word(v(v(sees)), sees).
```

Lösungen: DCGs

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

4.a) – c) Komplexe Nichtterminale

```
% ------------------------------------------------------------
%  det(Kasus, Numerus, Genus) - Artikel
% ------------------------------------------------------------
% Zwar erlaubt diese Grammatik gar keine Dativ- oder Genetiv-
% konstruktionen, dennoch sind die Artikel in allen Kasus
% aufgelistet. Im Sommersemester werden wir uns mit grösseren
% Grammatiken beschäftigen, wofür diese Grammatik als Ausgangs-
% punkt dienen wird.

det(nom, sg, m) --> [der].  % der Hund ist wichtig
det(gen, sg, m) --> [des].  % Max leidet wegen des Hundes
det(dat, sg, m) --> [dem].  % mit dem Hund ist das Leben schön
det(akk, sg, m) --> [den].  % Daniel dämonisierte den Hund

det(nom, pl, m) --> [die].  % die Hunde sind wichtig
det(gen, pl, m) --> [der].  % Max leidet wegen der Hunde
det(dat, pl, m) --> [den].  % mit den Hunden ist das Leben schön
det(akk, pl, m) --> [die].  % Daniel dämonisierte die Hunde

det(nom, sg, f) --> [die].  % die Tasse ist wichtig
det(gen, sg, f) --> [der].  % Max leidet wegen der Tasse
det(dat, sg, f) --> [der].  % mit der Tasse ist das Leben schön
det(akk, sg, f) --> [die].  % Daniel dämonisierte die Tasse

det(nom, pl, f) --> [die].  % die Tassen sind wichtig
det(gen, pl, f) --> [der].  % Max leidet wegen der Tassen
det(dat, pl, f) --> [den].  % mit den Tassen ist das Leben schön
det(akk, pl, f) --> [die].  % Daniel dämonisierte die Tassen

det(nom, sg, n) --> [das].  % das Haus ist wichtig
det(gen, sg, n) --> [des].  % Max leidet wegen des Hauses
det(dat, sg, n) --> [dem].  % mit dem Haus ist das Leben schön
det(akk, sg, n) --> [das].  % Daniel dämonisierte das Haus

det(nom, pl, n) --> [die].  % die Häuser sind wichtig
det(gen, pl, n) --> [der].  % Max leidet wegen der Häuser
det(dat, pl, n) --> [den].  % mit den Häusern ist das Leben schön
det(akk, pl, n) --> [die].  % Daniel dämonisierte die Häuser

% ------------------------------------------------------------
%  n(Kasus, Numerus, Genus) - Nomina
% ------------------------------------------------------------

n(nom, sg, m) --> [hund].   % der Hund geht
n(gen, sg, m) --> [hundes]. % Max leidet wegen des Hundes
n(dat, sg, m) --> [hund].   % mit dem Hund ist das Leben schön
n(dat, sg, m) --> [hunde].  % mit dem Hunde ist das Leben schön
n(akk, sg, m) --> [hund].   % Daniel dämonisierte den Hund
n(nom, pl, m) --> [hunde].  % die Hunde gehen
n(gen, pl, m) --> [hunde].  % Max leidet wegen der Hunde
n(dat, pl, m) --> [hunden]. % mit den Hunden ist das Leben schön
n(akk, pl, m) --> [hunde].  % Daniel dämonisierte die Hunde

n(nom, sg, f) --> [katze].  % der Hund geht
n(gen, sg, f) --> [katze].  % Max leidet wegen des Hundes
n(dat, sg, f) --> [katze].  % mit dem Hunde ist das Leben schön
n(akk, sg, f) --> [katze].  % Daniel dämonisierte den Hund
n(nom, pl, f) --> [katzen]. % die Katzen gehen
n(gen, pl, f) --> [katzen]. % Max leidet wegen der Katzen
n(dat, pl, f) --> [katzen]. % mit den Katzen ist das Leben schön
n(akk, pl, f) --> [katzen]. % Daniel dämonisierte die Katzen

% ------------------------------------------------------------
%  pron(Kasus, Person, Numerus) - Pronomina
% ------------------------------------------------------------
% Auch Wörter wie "mein" würden üblicherweise als Pronomina
% bezeichnet. Es sind auch nicht alle Kasus verzeichnet.

pron(nom, 1, sg) --> [ich].
```

```
pron(gen, 1, sg) --> [meiner].
pron(dat, 1, sg) --> [mir].
pron(akk, 1, sg) --> [mich].
pron(nom, 2, sg) --> [du].
pron(nom, 3, sg) --> [er].
pron(nom, 3, sg) --> [sie].
pron(nom, 3, sg) --> [es].
pron(nom, 1, pl) --> [wir].
pron(nom, 2, pl) --> [ihr].
pron(nom, 3, pl) --> [sie].

% -----------------------------------------------------------
%  v(Person, Numerus, Transitivität) - Verben
% -----------------------------------------------------------

v(1, sg, trans) --> [sehe].     % ich sehe den Hund
v(2, sg, trans) --> [siehst].   % du siehst den Hund
v(3, sg, trans) --> [sieht].    % Anna sieht den Hund
v(1, pl, trans) --> [sehen].    % wir sehen den Hund
v(2, pl, trans) --> [seht].     % ihr seht den Hund
v(3, pl, trans) --> [sehen].    % sie sehen den Hund

v(1, sg, intrans) --> [gehe].  % ich gehe
v(2, sg, intrans) --> [gehst]. % du gehst
v(3, sg, intrans) --> [geht].  % Anna geht
v(1, pl, intrans) --> [gehen]. % wir gehen
v(2, pl, intrans) --> [gehen]. % ihr geht
v(3, pl, intrans) --> [gehen]. % sie gehen

% -----------------------------------------------------------
%  s - Sätze
% -----------------------------------------------------------

s -->
  np(nom, Person, Numerus, _), % das Genus interessiert nicht
  vp(Person, Numerus).

% -----------------------------------------------------------
%  np(Kasus, Person, Numerus, Genus) - Nominalphrasen
% -----------------------------------------------------------

np(Kasus, 3, Numerus, Genus) -->
  det(Kasus, Numerus, Genus),  % der
  n(Kasus, Numerus, Genus).    % Katzen

np(Kasus, Person, Numerus, _) -->
  pron(Kasus, Person, Numerus).

% -----------------------------------------------------------
%  vp(Person, Numerus) - Verbalphrasen
% -----------------------------------------------------------

vp(Person, Numerus) -->
  v(Person, Numerus, intrans).

vp(Person, Numerus) -->       % [sie] sahen die Katze: Keine
  v(Person, Numerus, trans), % Kongruenz zwischen Verb und
  np(akk, _, _, _).          % Akkusativobjekt!
```

Von *s* ableitbare Terminalketten

der hund geht · der hund sieht den hund · der hund sieht die hunde · der hund sieht die katze · der hund sieht die katzen · der hund sieht mich · die hunde gehen · die hunde sehen den hund · die hunde sehen die hunde · die hunde sehen die katze · die hunde sehen die katzen · die hunde sehen mich · die katze geht · die katze sieht den hund · die katze sieht die hunde · die katze sieht die katze · die katze sieht die katzen · die katze sieht mich · die katzen gehen · die katzen sehen den hund · die katzen sehen die hunde · die katzen sehen die katze · die katzen sehen die katzen · die katzen sehen mich · ich gehe · ich sehe den hund · ich sehe die hunde · ich sehe die katze · ich sehe die katzen · ich sehe mich · du gehst · du siehst den hund · du siehst die hunde · du siehst die katze · du siehst die katzen · du siehst mich · er geht · er sieht den hund · er sieht die hunde · er sieht die katze · er sieht die katzen · er sieht mich · sie geht · sie sieht den hund · sie sieht die hunde · sie sieht die katze · sie sieht die katzen · sie sieht mich · es geht · es sieht den hund · es sieht die hunde · es sieht die katze · es sieht die katzen · es sieht mich · wir gehen · wir sehen den hund · wir sehen die hunde · wir sehen die katze · wir sehen die katzen · wir sehen mich · ihr gehen · ihr seht den hund · ihr seht die hunde · ihr seht die katze · ihr seht die katzen · ihr seht mich · sie gehen · sie sehen den hund · sie sehen die hunde · sie sehen die katze · sie sehen die katzen · sie sehen mich