Gedächtnisprotokoll Softwareentwicklung III - Logikprogrammierung 21.02.2007
Modul SE3 Wintersemester 06/07 - Prof. Wolfgang Menzel

Vorwort
Es gab insgesamt 60 Punkte zu erreichen, die erreichbaren Punkte für jede Aufgaben stehen in Klammern dahinter. Für die Bearbeitung der 6 Aufgaben waren insgesamt 2 Stunden veranschlagt. Vor jeder Teilaufgabe stehen die Punkte für diese.
Der Inhalt basiert auf der Vorlesung von Prof. Menzel zum Modul SE3 - Logikprogrammierung im Wintersemester 06/07. Alle Aufgaben wurden so oder in ähnlicher Weise im Übungsbetrieb behandelt oder zur Klausurvorbereitung in einer Probeklausur genannt. In der Klausur waren alle schriftlichen Hilfsmittel erlaubt, jedoch keine elektronischen wie etwa Taschenrechner, PDA, Handy oder Laptop.
Dieses Protokoll ist vollständig, allerdings nicht unbedingt komplett korrekt!

Aufgabe 1 (10P)

3 x 1P: a)

1. Was ist ein endrekursives Prädikat und woran erkennt man es?

Ein rekursives Prädikat ist ein Prädikat, dass sich selbst mit veränderten Argumenten wieder aufruft. Endrekursiv ist es, wenn dieser Aufruf das letzte Teilziel im Klauselkörper der Prädikatsdefinition(en) ist.

2. Welche Vorteile haben endrekursive Prädikate?

Da die vor dem rekursiven Aufruf berechneten Zwischenergebnisse nicht mehr benötigt werden, lassen sich endrekursive Prädikate iterativ ausführen – dies ist effizienter als eine Rekursion.

3. Unter welchen Bedingungen kann man für ein Berechnungsproblem eine endrekursive Prädikatsdefinition angeben?

Eine endrekursive Implementierung ist möglich, wenn die vor dem rekursiven Aufruf berechneten Zwischenergebnisse nicht in das Endergebnis eingehen, sich also das Prädikat so definieren lässt, dass das Ergebnis des rekursiven Aufrufs direkt dem Endergebnis entspricht.

5P: b)

```
mean(Liste_von_Zahlen,Mittelwert):-
        length(Liste_von_Zahlen,Laenge),
        sumlist(Liste_von_Zahlen,Summe),
        Mittelwert is Summe/Laenge.
```

Der Nachteil diese Prädikates ist, dass zwei rekursive Durchläufe durch die Liste notwendig sind.
Gesucht ist eine effizientere Implementierung, die die Liste nur einmal rekursiv abarbeitet. Geben Sie sowohl eine endrekursive als auch eine nicht-endrekursive Lösung an.

endrekursiv:

```
mean_er(List, Mean) :-
        mean_er(List, 0, 0, Mean).
mean_er([], Sum, Len, Mean) :-
        Mean is Sum / Len.
mean_er([N|List], Sum, Len, Mean) :-
        Sum2 is Sum + N,
        Len2 is Len + 1,
        mean_er(List, Sum2, Len2, Mean).
```

Nicht-endrekursiv:

```
mean_ner(List, Mean) :-
        sumlen(List, Sum, Len),
        Mean is Sum / Len.
sumlen([], 0, 0).
sumlen([N|List], Sum, Len) :-
        sumlen(List, Sum1, Len1),
        Sum is Sum1 + N,
        Len is Len1 + 1.
```

2P: c)
Welche Eigenschaften hat das Prädikat mean? Ankreuzen ob ja oder nein!
Symmetrisch, reflexiv, transitiv, 1:m, m:1
nein, nein, nein, nein, ja

Aufgabe 2 (10P)

Datenbasis:
a: %kunde(Kundennummer, Firmenname, Ort, Straße)
kunde(KNr, Name, Ort, Str)
b: %rechnung(Rechnungsnummer, Kundennummer, Datum, Produktgruppe, Betrag)
rechnung(RNr, Knr, Datum, Produktgruppe, Betrag)
c: %mahnung(Rechnungsnummer, Datum)
mahnung(RNr, Datum)
d: %zahlungseingang(Rechnungsnummer, Datum, Betrag)
zahlungseingang(RNr, Datum, Betrag)
Datum ist in Form von „2007/02/21“ gegeben. Ort als „12345-Musterstadt“

Schreiben Sie Anfragen mit so wenig überflüssigen Informationen wie möglich.

1P: a)
Welche Adresse hat die Firma „Scherzartikel-Mayer“?

```
kunde(_, 'Scherzartikel-Mayer', Ort, Straße).
```

2P: b)
An welche Kunden (Name + Ort) wurden im vergangenen Jahr Porzellanfiguren verkauft?
Hinweis: Der für die Datumsangaben verwendete Operatorenausdruck jjjj/mm/tt kann auch ungebundene Variabel enthalten!

```
rechnung(_, KNr, 2006/_/_, porzellanfiguren, _), kunde(KNr, Name, Ort, _).
```

3P: c)
Welche Firmen haben Ihre Rechnungen trotz Mahnung noch nicht bezahlt?
Hinweis: Die Negation einer Bedingung kann durch das Prädikat ' \+ ' /1 erfolgen.

```
rechnung(RNr, KNr, _, _, Gesamtbetrag),
mahnung(RNr, _),
findall(Betrag,
        zahlungseingang(RNr, _, Betrag),
        Betraege),
sumlist(Betraege, Bisher),
Bisher < Gesamtbetrag,
kunde(KNr, Name, Ort, _).
```

4P: d)
Eine 2. und letzte Mahnung wird verschickt, wenn eine Rechnung 4 Wochen nach der 1. Mahnung immer noch nicht bezahlt wurde. Wie oft ist dieser Fall im Jahr 2006 aufgetreten?

```
findall(RNr,
        (mahnung(RNr, Datum1),
         mahnung(RNr, Datum2),
        Datum1 @< Datum2,
        Datum2 = 2006/_/_),
        RNrs),
 length(RNrs, Anzahl).
```

Aufgabe 3 (10P)

2P: a)
Da Rechnungen auch teilweise bezahlt werden können, ist die Finanzabteilung daran interessiert, die tatsächlich noch zu erwartenden Zahlungen zu ermitteln.
Definieren Sie ein Prädikat, dass für eine gegebene Rechnung den noch ausstehenden Betrag ermittelt.

```
offen(RNr, Restbetrag) :-
        rechnung(RNr, _, _, _, Betrag),
        findall(Eing,
                zahlungseingang(RNr, _, Eing),
                Eingang),
        sumlist(Eingang, Bisher),
        Restbetrag is Betrag - Bisher.
```

4P: b)
Definieren sie ein Prädikat, dass für eine gegebene Kundennummer die Summe der noch offenen Rechnungen ermittelt.

```
schulden(KNr, Schulden) :-
        findall(Offen,
                (rechnung(RNr, KNr, _, _, _),
                 offen(RNr, Offen)),
                Schuld),
        sumlist(Schuld, Schulden).
```

4P: c)
Die Firmenleitung möchte wissen, welche Kunden die höchsten Außenstände haben.
Gegeben ist eine Liste, in der die Kunden und Ihre Außenstände als Dreiertupel [Firma, Ort, Betrag] kodiert sind.
Definieren Sie ein Prädikat, dass aus der Liste eine Rangliste der höchsten Außenstände ermittelt. Die Länge der Rangliste muss frei wählbar sein und die Einträge sind nach fallender Summe der Verbindlichkeiten zu sortieren.
Hinweis: Liste umformen, sodass mit Standardwerkzeugen sortiert werden kann (sort/2, setof/3 ...).
Beachten, dass bei Sortierung numerische Daten in aufsteigender Reihenfolge angeordnet werden. Alternativ dazu können Sie auch ein bekanntes Sortierprädikat modifizieren.

```
rangliste(Eingabe, Laenge, Rangliste) :-
        umbauen(Eingabe, Eing2),
        sort(Eing2, Eing3),
        reverse(Eing3, Eing4),
        praefix(Eing4, Laenge, Rangliste).
umbauen([], []).
umbauen([[Firma, Ort, Betrag]|L1], [[Betrag, Firma, Ort]|L2]) :-
        umbauen(L1, L2).
praefix([], _, []) :- !.
praefix(_, 0, []) :- !.
praefix([A|L], N, [A|L2]) :-
        N2 is N-1,
        praefix(L, N2, L2).
```

Aufgabe 4 (10P)

2P: a)

```
umsatz_pro_kunde_und_jahr1(KNr, Jahr, Umsatz) :-
        kunde(KNr,_,_,_),
        findall(ZBetrag,
                rechnung(RNr,KNr,_,_,_),
                        zahlungseingang(RNr,Jahr/_/_,ZBetrag)),
                Betraege),
        sumlist(Betraege,Umsatz).

umsatz_pro_kunde_und_jahr2(KNr, Jahr, Umsatz) :-
        findall(ZBetrag,
                rechnung(RNr,KNr,_,_,_),
                        zahlungseingang(RNr,Jahr/_/_,ZBetrag)),
                Betraege),
        sumlist(Betraege,Umsatz).
```

Hier habe ich die Fragestellung vergessen, vermute aber, dass so was wie „Zeigen Sie den Unterschied zwischen den beiden Prädikaten auf“ gefragt wurde ;-) (Worin liegt der Unterschied im Verhalten der beiden Prädikate?)
Das erste Prädikat prüft vorher, ob die angegebene Kundennummer überhaupt existiert - wenn nicht, schlägt es fehl, während das zweite Prädikat einfach den Umsatz 0 zurückliefern würde.
Im Falle eines Aufrufs mit uninstanziiertem erstem Argument liefert das erste Prädikat die Umsätze aller Kunden als Alternativen, während das zweite Prädikat alle Umsätze aufaddieren würde {wirklich? oder liefert es den Umsatz des Kunden, der zufällig als erstes gefunden wurde?}.

2P: b)
Die Firmenleitung möchte wissen, wie sich die Kunden hinsichtlich der Umsatzstärke verteilen. Dazu teilen Sie die Kunden in Gruppen mit unterschiedlichen Umsatzverhalten auf. Die Gruppen werden durch eine Liste der Intervallgrenzen (in EURO) definiert, z. B.: [0,10,100,100,10000,.....].
Schreiben Sie ein Prädikat, dass die Anzahl der Kunden in einem gegebene Intervall für 2006 bestimmt.

```
umsatz_zwischen(Min, Max, Anzahl) :-
        findall(Umsatz,
                (umsatz_pro_kunde_und_jahr1(_, 2006, Umsatz),
                 Min =< Umsatz,
                 Umsatz < Max),
                Umsaetze),
        length(Umsaetze, Anzahl).
```

2P: c)
Definieren Sie ein Prädikat, dass die Anzahl der Kunden in allen Intervallen einer gegebene Liste von Intervallgrenzen ermittelt und diese wiederum in einer Liste zusammenfasst.
Hinweis: Arbeiten Sie dazu die Liste der Intervallgrenzen rekursiv ab.

```
umsatzgruppen([], []).
umsatzgruppen([_], []).
umsatzgruppen([Min, Max|Intervalle], [Anzahl|Anzahlen]) :-
        umsatz_zwischen(Min, Max, Anzahl),
        umsatzgruppen([Max|Intervalle], Anzahlen).
```

2P: d)
Resultat von c in Liste absoluter Häufigkeiten für die jeweiligen Umsatzgruppen. Für eine diskrete Wahrscheinlichkeitsverteilung muss zusätzlich die Bedingung gelten, dass sich alle Einzelwerte der Verteilung zu Eins addieren.
Definieren Sie ein Prädikat, dass eine gegebene Verteilung in eine Wahrscheinlichkeitsverteilung umwandelt.

```
normalisieren(Eingabe, Ausgabe) :-
        sumlist(Eingabe, Summe),
        normalisieren(Eingabe, Summe, Ausgabe).
normalisieren([], _, []).
normalisieren([NE|E], S, [NA|A]) :-
        NA is NE / S,
        normalisieren(E, S, A).
```

2P: e)
Eine unimodale Verteilung besitzt genau einen Maximalwert.
Definieren Sie ein Prädikat, dass überprüft, ob eine gegebene Verteilung unimodal ist.

```
unimodal([]).           %die leere Liste ist immer unimodal
unimodal([_]).          %eine ein-elementige Liste auch
unimodal(L) :-          %sortieren und gucken, ob die beiden größten Elemente unterschiedlich sind
        msort(L, L2),
        reverse(L2, L3),
        [A, B|_] = L3,
        A > B.
```

Aufgabe 5 (8P)

5 x 1P: a)
Unifizieren Sie die folgenden Ausdrücke (falls möglich) und geben Sie die dabei erzeugten Variablenbindungen an.

Ausdruck	Erfolgreich	Variablenbindung
add(3,6,Summe) add(Smd1, Smd2, 7)	ja	Smd1=3, Smd2=6, Summe = 7
g(g(g(D))) g(g(g(g,E)))	nein	
x([F,r,q]) x([[G,H],H|G])	ja	F = [[q],r] H = r G = [q]
a(2 * 4 + 3) a(2 * B)	nein	
[a,b,c|X] /X [a,b,c,d|Y]/[d|Y]	ja	X = [d|Y]

3P: b)
Wozu dienen Variablen in der relationalen Programmierung?
Unifikation, Übergabe von Argumenten an Prädikate und Aufnahme von Ergebnissen zur Weiterverarbeitung.

An welchen Stellen in einem Programm können Variablen benutzt werden?
Rein syntaktisch überall dort, wo auch ein Term stehen kann.

Was ist eine Variablenbindung und wie wird sie hergestellt bzw. gelöst?
Hergestellt wird eine Variablenbindung durch Unifikation, gelöst werden kann sie nicht mehr.

Aufgabe 6 (12P ???)

1P: a)
Wozu evaluiert der folgende Scheme-Ausdruck?

```
(map (lambda (x) (-  0  x))
        quote (1  2  3))
```

(-1 -2 -3)

?P: b) Zu welchen der folgenden Scheme-Ausdrücke ist der oben angegebene Scheme-Ausdruck bedeutungsgleich?

1. (map (compose zero? -) (quote (1 2 3)))
2. (map (lambda (x) (+ x 0)) (quote (1 2 3)))
3. (map (curry * -1) (quote (1 2 3)))
4. (map (lamda (x) (- x)) (quote (1 2 3)))
5. (filter negative? (quote (1 2 3)))
6. (map negative? (quote (1 2 3)))

Nur zu 3. und 4.

2P: c)
Was bewirkt die special form expression quote und warum braucht man sie?
Eine Liste wie (a b c) wird normalerweise zum Ergebnis des Funktionsaufrufs a mit den Argumenten b und c ausgewertet. Möchte man eine Liste beispielsweise als Argument an eine Funktion übergeben, muss man kennzeichnen, dass es sich dabei um keinen Funktionsaufruf handelt, z.B. (quote (a b c)) oder kurz '(a b c).

1P: d)
Erklären Sie, was die folgende Funktion berechnet:

```
;; x ist eine Liste von Zahlen
(define (berechne x),
        (if (null? x)
                0
                (+ (car x) (berechne (cdr x)))
        )
)
```

Die Funktion berechnet die Summe einer Liste von Zahlen.

?P: e)
Geben Sie für die Funktion aus Aufgabenteil d) eine entsprechende Prädikatsdefinition in Prolog.

```
berechne([], 0).
berechne([Car|Cdr], S) :-
        berechne(Cdr, S1),
        S is Car + S1.
```

?P: f) Diskutieren Sie eventuelle Unterschiede im Verhalten des Programms aus Aufgabenteil d) und e).
Abgesehen davon, dass d) ein Ergebnis zurückliefert und e) das Ergebnis in eine zusätzliche Argumentposition hineinunifiziert, lässt sich das Prolog-Prädikat auch direkt als Testprädikat benutzen (z.B. berechne([4,1], 5)).