Probe-. Klausur zu 'Softwareentwicklung III:
Funktionale Programmierung'
WS 2009/2010
Leonie Dreschler-Fischer

Note:_____ und Begründung:

Hamburg,_____ Unterschrift (Prüferin)_____

Rechtsmittelbelehrung: Gegen die Bewertung dieser Prüfungsleistung kann innerhalb eines Monats nach ihrer Bekanntgabe Widerspruch erhoben werden. In diesem Zeitraum kann die Bewertung der Klausur eingesehen werden. Der Widerspruch ist schriftlich oder zur Niederschrift bei der bzw. dem Vorsitzenden des für das Hauptfach zuständigen Prüfungsausschusses einzulegen. Es wird darauf hingewiesen, dass ein erfolgloses Widerspruchsverfahren kostenpflichtig ist.

Zugelassene Hilfsmittel: Die ausgeteilten Prüfungsunterlagen, ein Handbuch zu Scheme (Revised Report on Scheme), ein Wörterbuch, Schreibstifte.

Nicht verwendet werden dürfen: Mobiltelefone oder sonstige elektronische Kommunikationsmittel, sonstige schriftliche Aufzeichnungen zur Vorlesung und zu den Übungen.

Datum & Unterschrift Klausurteilnehmer/in: _____

Wichtige Hinweise:

- Die Klausur bitte auf jeden Fall geheftet lassen! Kein zusätzliches Papier verwenden!
- **Unterschreiben Sie die Klausur auf dieser Seite.**
- Punkte für Teilaufgaben werden durch $\{\oslash\}$ angegeben, z.B. $\{\oslash\,\oslash\}$ = 2 Punkte. Die Gesamtanzahl von Punkten pro Aufgabe steht jeweils im Kasten am Rand der entsprechenden Aufgabe unter "von". Beachten Sie, dass Sie für jeden zu erzielenden Punkt etwa zwei Minuten Zeit zur Verfügung haben.
- Sollte der Platz für eine Aufgabe nicht ausreichen, so verwenden Sie die Leerseiten am Ende und machen Sie einen Vermerk am Rand, etwa siehe "$\Longrightarrow$ Seite **??**".
- Für alle Aufgaben können Sie die Funktionen aus dem tools-module der se3-bib als vordefiniert voraussetzen.

```
#lang scheme
(require "se3-bib/tools-module.ss")
```

- Im Anhang der Klausur, auf Seite **??**, finden Sie ein kleines Handbuch zu Funktionen höherer Ordnung, CLOS und Prolog-in-Scheme.

1 Aufgaben

1. Zu welchen Werten evaluieren die folgenden Scheme-Ausdrücke?

   (a) `(+ 3 (- 6 7))` {⊘}

   Lösung: 2

   (b) `'(+ 3 (- 6 7))` {⊘}

   Lösung: (+ 3 (- 6 7))

   (c) `(car '(auto bus))` {⊘}

   Lösung: auto

   (d) `(cdr '(auto bus))` {⊘}

   Lösung: (bus)

   (e) `(map sqrt '(1 9 4 25))` {⊘⊘}

   Lösung: `(1 3 2 5)`

   (f) `((curry < 6) 2)` {⊘⊘}

   Lösung: `#f`

   (g) `(reduce * '(1 2 6 0 1) 1)` {⊘⊘}

   Lösung: 0

von	
10	

von
10

2. Reduktionsstrategien

(a) Geben Sie einen Scheme-Ausdruck an, für den die Auswertung durch innere Reduktion vorteilhafter ist als die äußere Reduktion. Begründung? {⊘⊘}

Lösung:

```
(define (square x)(* x))

(square (square (+ 1 2)))
```

`(+ 1 2)` wird bei innerer Reduktion nur einmal ausgewertet, bei aeußerer viermal.

(b) Ausdrücke mit `if` und `cond` werden in Scheme anders ausgewertet als funktionale Ausdrücke. {⊘⊘⊘}
Worin besteht der Unterschied, und warum ist das notwendig?
Nennen Sie zwei weitere *special form operators*.

Losung: if, cond sind special form operators. Jeder special form Operator bestimmt die Reduktionsreihenfolge der Argumente einer special form expression.
Nennen Sie zwei weitere special form operators: `set`, lambda, define, and, or !

(c) Zur Syntax von Scheme: {⊘⊘⊘⊘⊘}
Gegeben seien die folgenden Definitionen:

```
(define *a* 10)
(define *b* '*a*)
(define (merke x)(lambda () x))
(define (test x)
 (let ((x (+ x *a*))))
 (+ x 2))
```

Haben die folgenden Ausdrücke einen wohldefinierten Wert und zu welchen Werten evaluieren sie?

1. `*a*`

2. `(+ *a* *b*)`

3. `(+ (eval *a*) (eval *b*))`

4. `(and (> *a* 10) (> *b* 3))`

5. `(or (> *a* 10) (/ *a* 0))`

6. `(+ 2 (merke 3))`

7. `(+ 2 ((merke 3)))`

8. `(test 4)`

von
10

3. **Formen der Rekursion:** Gegeben seien die folgenden Funktionsdefinitionen:

```
(define (alle-neune xs)
   (cond ((null? xs) 0)
         ((= 9 (car xs)) (+ 1 (alle-neune (cdr xs))))
         (else (alle-neune (cdr xs)))))

(define (alle-zehne  xs wieviele)
   (cond ((null? xs) wieviele)
         ((= 10 (car xs))
          (alle-zehne (cdr xs) (+ 1 wieviele)))
         (else (alle-zehne (cdr xs) wieviele))))

(define (anzahl-atome xs)
  (cond ((null? xs) 0)
        ((atom? xs) 1)
        (else (+ (anzahl-atome (car xs))
                 (anzahl-atome (cdr xs))))))
```

Betrachten Sie die angegebenen Funktionen `alle-neune, alle-zehne` und `anzahl-atome`. Geben Sie für jede dieser Funktionen an, welche Art von Rekursion vorliegt. {⊘⊘⊘⊘⊘}

Begründen Sie Ihre Antwort. {⊘⊘⊘⊘⊘}

Lösung:

alle-neune: linear rekursiv - in jeder Variante wird die Funktion hoechstens einmal rekursiv verwendet.

nicht endrekursiv

alle-zehne: linear rekursiv und endrekursiv

anzahl-atome: baumrekursiv, im else-Zweig wird die Funktion zweimal rekursiv

4. **Rekursion:**

von
10

(a) Nennen Sie eine rekursive Scheme-Datenstruktur. Begründung? {⊘⊘}
Lösung: Eine Liste ist eine rekursive Datenstruktur, die entweder eine leere Liste ist oder ein Kopfelement, gefolgt von einer Liste.

(b) Definieren Sie eine *rekursive* Funktion `(laengen xss)` in Scheme, die folgendes leistet:
Gegeben sei eine Liste von Listen. Bilden Sie diese ab auf die Liste der Längen der Unterlisten.
Beispiel:
`(laengen (1 3 4) (Auto Bus) () (3 4 5 6))` ⟶ `(3 2 0 4)`.

Geben Sie die Funktion in drei Varianten an:

i. linear rekursiv, {⊘⊘}
Lösung:

```
(define (laengen xs)
  (if (null? xs) '()
   (cons (length (car xs))
         (laengen (cdr xs)))))
```

ii. endrekursiv, {⊘⊘}
Lösung:

```
(define (laengen-akk akk xs)
  (if (null? xs) (reverse akk)
   (laengen-akk
    (cons (length (car xs))
    akk)(cdr xs))))
```

iii. und mittel einer Funktion höherer Ordnung {⊘⊘}
Lösung:

```
(define (laengenHigh xss)
  (map length xss))
```

(c) Woran erkennen Sie eine endrekursive Funktion?{⊘⊘}
Lösung: Daran, daß das Resultat des rekursiven Aufrufs direkt als Wert zurueckgegeben wird. Die Verwendung eines Akkumulators allein ist weder ein notwendiges noch ein hinreichendes Kriterium.

von
14

5. **Funktionen höherer Ordnung: Regressionsgerade**
Gegeben sei eine Liste von Wertpaaren `((`$x_1.y_1$`),(`$x_2.y_2$`),...(`$x_n.y_n$`))` als gepunk-tete Paare. Definieren Sie Scheme-Funktionen zur Berechnung der folgenden Wer-te. Verwenden Sie dabei nach Möglichkeit Funktionen höherer Ordnung, wie `map`, `reduce`, `curry` usw.

(a) Eine Funktion `xliste`, die aus einer Liste von Wertpaaren die Liste aller x-Komponenten extrahiert, z.B. {⊘⊘}

`(xliste '((1 . 3) (2 . 4) (3 . 5)))` ⟶ `(1 2 3)`

Lösung:

```
(define (xliste pairs
  (map car pairs))
```

(b) Eine Funktion `yliste`, die aus einer Liste von Wertpaaren die Liste aller y-Komponenten extrahiert, z.B. {⊘⊘}

`(yliste '((1 . 3) (2 . 4) (3 . 5)))` ⟶ `(3 4 5)`

Lösung:

```
  (map cdr pairs))
```

(c) Eine Funktion `xsumme`, die die x-Werte einer Liste von Wertpaaren addiert $\sum_{i=1}^{n} x_i$: {⊘⊘}

`(xsumme '((1 . 3) (2 . 4) (3 . 5)))` ⟶ `6`

Lösung:

```
(define (xsumme pairs)
 (apply + (xliste pairs)))
(define (ysumme pairs)
 (apply + (yliste pairs)))
```

(d) Eine Funktion `x*y-Summe`, die die xy-Produkte einer Liste von Wertpaaren addiert $\sum_{i=1}^{n} x_i \cdot y_i$: {⊘⊘}

`(x*y-Summe '((1 . 3) (2 . 4) (3 . 5)))` ⟶ `26` ¨

Lösung:

```
(define (x*y-summe pairs)
  (apply +
   (map * (xliste pairs)
          (yliste pairs))))
```

(e) Eine Funktion `x**2-Summe`, die die Quadrate der x-Werte einer Liste von Wertpaaren addiert $\sum_{i=1}^{n} x_i^2$: {⊘⊘}

`(x**2-Summe '((1 . 3) (2 . 4) (3 . 5)))` ⟶ `14`

Lösung:

```
(define (x**2-summe pairs)
  (apply +
    (map (lambda (x) (* x x))
         (xliste pairs))))
```

(f) Die Gerade $y = m \cdot x + c$, deren quadratischer Anpassungsfehler für eine gegebene Menge von Wertpaaren $\{(x_1, y_1), \cdots, (x_n, y_n)\}$ am kleinsten ist, hat die Steigung m und den Achsenabschnitt c mit den folgenden Werten:

$$\begin{aligned} m &= \frac{n \cdot \sum_{i=1}^{n}(x_i \cdot y_i) - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i}{n \cdot \sum_{i=1}^{n} x^2 - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} x_i} \\ c &= \frac{\sum_{i=1}^{n} y_i - m \cdot \sum_{i=1}^{n} x_i}{n} \end{aligned}$$

Zusatzaufgabe: Definieren Sie eine Scheme-Funktion `anpassungsgerade` als Funktion höherer Ordnung, die für eine Menge von Wertpaaren die Anpassungsgerade berechnet und eine *Scheme-Funktion* als Wert zurückgibt, die die y-Werte der Geraden als Funktion von x berechnet. Beispiel: {⊘⊘⊘⊘}

```
> (define g-anp
     (anpassungsgerade '((1 . 0.5) (2 . 1) (3 . 1.5))))
> (g-anp 2)  -> 1
```

denn die optimale Anpassungsgerade ist die Gerade $y = 0.5 \cdot x$, und der y-Wert für $x = 2$ ist 1.

6. **Vererbung in CLOS:**

(a) Definieren Sie als **CLOS-Klasse** eine Klasse von Fahrzeugen und spezialisie-ren Sie diese Klassen für unterschiedliche Medien, in denen sich die Fahrzeu-ge bewegen: Bodengebundene Landfahrzeuge, schwimmfähige Wasserfahrzeuge, flugfähige Luftfahrzeuge. {⊘ ⊘ ⊘} **Lösung:**

```
(defclass Fahrzeug ()

  (passagierzahl
   :initarg :passagierzahl
  :reader passagierzahl)
  :printer #t
  :documentation "eine␣Klasse␣von␣Fahrzeugen")

(defclass Landfahrzeug (Fahrzeug)
  (medium-land
   :initvalue 'Land :reader
  medium)(v-max-land
   :initarg :v-max-land
  :reader v-max)
  (zuladung-land
   :initarg :zuladung-
   land:reader zuladung)
  (verbrauch-land
   :initarg :verbrauch-land
  :reader verbrauch)
  :documentation "Bodengebundene␣Landfahrzeuge")

(defclass Wasserfahrzeug (Fahrzeug)
  (medium-wasser
   :initvalue 'Wasser :reader medium)
  (v-max-wasser
   :initarg :v-max-wasser :reader v-max)
  (zuladung-wasser
   :initarg :zuladung-wasser :reader zuladung)
  (verbrauch-wasser
   :initarg :verbrauch-wasser :reader verbrauch)
  :documentation "schwimmfähige␣Wasserfahrzeuge")
```

```
 :initvalue 'Luft :reader medium)
(v-max-luft
 :initarg :v-max-luft :reader v-max)
(zuladung-luft
 :initarg :zuladung-luft :reader zuladung)
(verbrauch-luft
 :initarg :verbrauch-luft :reader
verbrauch):documentation "flugfähige␣
Luftfahrzeuge")
```

(b) Definieren Sie Klassen von Mehrzweckfahrzeugen: {⊘⊘}

i. Ein Amphibienfahrzeug, das sich zu Wasser und zu Lande bewegen kann.

ii. Ein Batman-Superfahrzeug, das sich zu Wasser, zu Lande und in der Luft bewegen kann. **Lösung:**

```
(defclass Amphibienfahrzeug (Wasserfahrzeug Landfahrzeug)
  :documentation
  "kann␣sich␣zu␣Wasser␣und␣zu␣Lande␣bewegen")
```

```
(defclass Batmobil (Wasserfahrzeug Landfahrzeug Luftfahrzeug)
     :documentation
     "kann␣sich␣zu␣Wasser,␣zu␣Lande␣und␣in␣der␣Luft␣bewegen")
```

(c) Die Klasse Fahrzeug soll folgende Operationen bieten: {⊘⊘⊘⊘⊘⊘}

i. Abfrage des Mediums, in dem sich das Fahrzeug bewegt,

ii. Abfrage der Maximalgeschwindigkeit,

iii. Abfrage der Zuladung (Tragfähigkeit)

iv. Abfrage des Verbrauchs pro 100 km

v. Abfrage der der Passagierzahl

Spezifizieren Sie generische Funktionen als Signatur für die Operationen der Klasse Fahrzeug und diskutieren Sie, welche Methodenkombination für die jeweilige Operation sinnvoll erscheint.

Lösung:

Das Medium: Hier ist die `generic-list-combination` sinnvoll, damit alle Medien in einer Liste angegeben werden.

```
(defgeneric medium ((Fahrzeug))
  :combination generic-list-combination)
```

Maximalgeschwindigkeit: Hier ist die `generic-max-combination` sinnvoll, da die Maximalgeschwindigkeit des Fahrzeugs eben die maximale Geschwin-digkeit überhaupt ist.

```
(defgeneric v-max ((Fahrzeug))
        :combination generic-max-combination)
```

Zuladung: Hier ist die `generic-min-combination` sinnvoll, um den sicheren Betrieb des Fahrzeugs in allen Medien zu gewährleisten.`(defgeneric`

```
zuladung ((Fahrzeug))
        :combination generic-min-combination)
```

Verbrauch: Hier ist eine Methodenkombination für den Durchschnitt sinnvoll, da es sich bei einer Verbrauchsangabe meist um den Durchschnittsverbrauch handelt. Dies müßte man aber selbst definieren. Will man mit den vordefinierten Methodenkombinationen auskommen, soltte man als Käufer die max-Kombination wählen, um konservativ mit dem schlimmsten Fall zu rechnen, aber als Verkäufer die min-Kombination bevorzugen, um mit der Bestleistung zu werben.

```
(defgeneric verbrauch ((Fahrzeug))
  :combination generic-max-combination)
```

Eine eigene Methodenkombination kann man in CLOS so definieren:

```
(define generic-avg-combination
  (make-generic-combination
   :init '(0 . 0)
   :combine (lambda (x acc)
              (cons (+ x (car acc))
                    (+ 1 (cdr acc))))
        :process-result (lambda (res)
                   (/ (car res)
                      (cdr res)))))

   (defgeneric verbrauch ((Fahrzeug))
:combination generic-avg-combination)
```

Passagierzahl: Hier kann auf man auf die Angabe einer Methodenkombination verzichten, da ein einziger Slot für die Passagierzahl unabhängig vom Medium sinnvoll ist, sofern die zugelassene Passagierzahl nur von der Bauform des Fahrzeugs und nicht vom Medium und der Tragkraft abhängt.

```
(defgeneric passagierzahl ((Fahrzeug)))
```

von
10

7. **Prolog in Scheme:**

Gegeben sei eine Prolog-in-Scheme-Datenbasis, die Angaben zu Medikamenten, deren Wirkstoffen und Indikationen enthält.

- Die Relation „wirkstoff“ beschreibt, welchen Wirkstoff ein Medikament enthält, z.B. Acetylsalicylsäure.
- Die Relation „Indikation“ beschreibt, für welche Krankheiten das Medikament eingesetzt werden kann.

Ein Auszug aus der Datenbasis:

```
;(wirkstoff Medikament Wirkstoff)
(<- (wirkstoff "Aspirin_Plus_C" Acetylsalicylsäure))
(<- (wirkstoff "Aspirin_Plus_C" Vitamin_C))
(<- (wirkstoff "Aspro" Acetylsalicylsäure))
(<- (wirkstoff "Spalt" Acetylsalicylsäure))
(<- (wirkstoff "Paracetamol-Ratiopharm" Paracetamol))
(<- (wirkstoff "Talvosilen" Paracetamol))
(<- (wirkstoff "Talvosilen" Codein))
(<- (wirkstoff "Proviron" Mesterolon))
(<- (wirkstoff "Andriol" Testetoronundecanoat))

;(indikation Wirkstoff Anwendung)
(<- (indikation Acetylsalicylsäure
                "Fiebersenkend"))
(<- (indikation Acetylsalicylsäure
                "Schmerzmittel"))
(<- (indikation Codein "Hustendaempfer"))
(<- (indikation Mesterolon "Hormonmangel"))
```

Formulieren Sie die folgenden Fragen in Prolog-in-Scheme, und stellen Sie entsprechende Anfragen an die Datenbasis:

(a) Welche Wirkstoffe enthält das Medikament „Aspirin Plus C“? {⊘}

Lösung:

```
(?- (wirkstoff "Aspirin_Plus_C" ?Wirkstoff))
```

(b) Welche Wirkstoffe haben dieselben Anwendungsgebiete wie der Wirkstoff „Acetylsalicylsäure“? {⊘⊘}

Lösung:

```
(?- (indikation Acetylsalicylsäure  ?indik)
    (indikation ?w2 ?indik)
    (!= ?w2 Acetylsalicylsäure))
```

(c) Welche Medikamente sind Hustendämpfer? {⊘⊘}

Lösung:

```
(?- (indikation ?wirkstoff "Hustendaempfer")
      (wirkstoff ?medik ?wirkstoff))
```

(d) Welche Medikamente sind als Kombinationspräparate bedenklich, da sie mehr als einen Wirkstoff enthalten? {⊘⊘}

Lösung:

```
(?- (wirkstoff ?Medikament ?Wirkstoff1)
    (wirkstoff ?Medikament ?
    Wirkstoff2)(!= ?Wirkstoff1 ?
    Wirkstoff2))
```

(e) In den Sprachen Prolog und PrologInScheme sind funktionale Sprachelemente eingebettet. Nennen Sie ein Beispiel: {⊘}
Lösung: Das Prädikat is“ bindet die Werte funktionaler Ausdrücke an ble. Varia-”

(f) Was ist zu beachten, wenn in relationalen Programmiersprachen funktionale Sprachelemente verwendet werden? Welche wichtige Eigenschaft geht verloren?{⊘⊘}
Lösung: Alle Variablen der funktionalen Ausdrücke müssen vorher instanziiert sein. Die Richtungsunabhängigkeit geht verloren.

2 Funktionslexikon

2.1 Funktionen höherer Ordnung

2.1.1 Verknüpfung von Funktionen

(curry f arg1 ... argn) : Partielle Anwendung von `f` auf `arg1 ... argn` von links nach rechts

(rcurry f argm ... argn) : Partielle Anwendung von `f`, Bindung der Argumente von rechts nach links

(compose f1 ... fn) : Funktionskomposition, Hintereinanderausführung
`(f1 (... (fn x)))`

(conjoin p1? ... pn?) : Konjunktion von Prädikaten

(disjoin p1? ... pn?) : Disjunktion von Prädikaten

(always x) : Die konstante Funktion, deren Wert unabhängig von den Argumenten `x` ist.

2.1.2 Idiome der funktionalen Programmierung

(apply f xs) : Anwendung einer Funktion `f` auf eine Liste von Argumenten `xs`

(map f xs1 ... xsn) : Abbilden einer oder mehrerer Listen auf die Liste der Bilder der Elemente. Die Stelligkeit der Funktion f muß mit der Anzahl der angegebenen Listen xs1...xsn usw. übereinstimmen.

(filter p? xs) : Die Liste der Elemente von `xs`, die `p?` erfüllen

(reduce f xs seed) : Paarweise Verknüpfung der Elemente von `xs` mit `f`, Startwert `seed`,
`(f x1 (f x2 (... (f xn seed))))`

(iterate f end? start) : Die Liste der Funktionsanwendungen, bis `end?` erfüllt ist,
`(start (f start) (f (f start)) ...)`

(untilM f end? start) : Der erste Wert der Folge `start, (f start), (f (f start)), ...` der `end?` erfüllt

(some p? xs) : Finde das erste Element von `xs`, das das Prädikat `p?` erfüllt, ansonsten gebe #f zurück.

(every p? xs) : wahr, wenn alle Elemente von `xs` das Prädikat `p?` erfüllen.

(assoc key alist) : Suche in der Assoziationsliste `alist` das erste Paar, daß das als *Kopf* den Schlüssel `key` enthält.

(rassoc key alist) : Suche in der Assoziationsliste `alist` das erste Paar, daß das als *Rest* den Schlüssel `key` enthält.

2.2 CLOS

2.2.1 Klassen

```
(defclass <Name der Klasse> ({<Oberklassen>})
  {(<Slot> {<Slot-keys>})}
  {<Class-keys>}
  )
```

Optionen für die Attribute (slots):

- `:initarg keyword: <key>` *Schlüsselwort für den Konstruktor*
- `:initializer <func>` *Initialisierungsfunktion*
- `:initvalue <value>` *Defaultwert*
- `:reader <name>` *Akzessorfunktion zum Lesen des Wertes*
- `:writer <name>` *Akzessorfunktion zum Schreiben des Wertes*
- `:accessor <name>` *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- `:accessor <name>` *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- `:type <type>` *Typdefinition für die Klasse des slots*

```
(defgeneric <name>({(<arg> <class>)} {<arg>})
    {:combination <combination>}
    {:documentation <string>} )

(defmethod <name> [<qualifier>]({(<arg> <class>)} {<arg>})
    {:documentation <string>} )
```

2.2.2 Ergänzungsmethoden

```
<qualifier> ::= :after | :before | :around
```

2.2.3 Methodenkombinationen

```
generic-+-combination, generic-list-combination
generic-min-combination, generic-max-combination
generic-append-combination, generic-append!-combination
generic-begin-combination, generic-and-combination
generic-or-combination
```

2.3 Prolog-in-Scheme-Lexikon

Variable: ?X : Die Namen von Variablen beginnen mit einem Fragezeichen.

Anonyme Variable: ? : Ein Fragezeichen bezeichnet eine anonyme Variable, die in Ausgaben unterdrückt wird.

Regeln: Klauseln mit Prämissen (Zielen) :

```
<Klausel-Kopf> :- <Ziel 1>...<Ziel n>
```

Negation: not : Der not-Oprator negiert eine Klausel : `(not <Klausel>)`

Ungleichheit: != : Das !=-Prädikat ist wahr, wenn zwei Strukturen oder Variablen nicht unifizieren :

```
(!= <struktur1> <struktur2> )
```

Das assert-Macro: ← : Trage eine Klausel in die Datenbasis ein.

```
(<- <Klausel>)
```

Das query-Macro: ?- : Anfrage, durch welche Variablenbindungen die Konjunktion der Ziele in der Anfrage erfüllt werden kann.

```
(?- <Ziel 1> ... <Ziel n>)
```

findall: findall sammelt alle Resultate des Prädikatsaufrufs `<Ziel>` in einer Liste `<Liste>` als instanziierte Varianten des Ausdrucks `<Term>` .

```
(findall <Term> <Ziel> <Liste>)
```

count: count zählt die Resultate des Prädikatsaufrufs `<Ziel>` und bindet die Anzahl an die Variable `<Var>`.

```
(count <Var> <Ziel>)
```

Funktionale Auswertung: is: Das `is`-Prädikat bindet den Wert eines funktionalen Ausdrucks an eine Variable. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

```
(is <Var> <Ausdruck> )
```

Funktionale Auswertung: test: Das `test`-Prädikat evaluiert einen funktionalen Ausdruck. Das Prädikat ist erfüllt, wenn der Ausdruck wahr ist. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

```
(test  <Ausdruck> )
```

Zusätzliche Leerseiten

Aufgabe	Punkte
1	
2	
3	
4	
5	
6	
7	
Summe:	
von	75
Bestanden?	**Ja** ◯ **Nein** ◯

Datum & Unterschrift Vorkorrektur _____

Datum & Unterschrift Prüferin _____