Hinweise zur Bearbeitung der Klausur zum Kurs 1816 “Logisches und funktionales Programmieren”

Wir begrüßen Sie zur Klausur “Logisches und funktionales Programmieren”. Bitte lesen Sie sich diese Hinweise vollständig und aufmerksam durch, bevor Sie mit der Bearbeitung beginnen:

1. Prüfen Sie bitte die Vollständigkeit Ihrer Unterlagen. Die Klausur umfaßt:
   - 2 Deckblätter,
   - diese Hinweise zur Bearbeitung,
   - 1 Formblatt für eine Bescheinigung für das Finanzamt:
   - 7 Aufgaben.
2. Bitte füllen Sie, bevor Sie mit der Bearbeitung der Klausuraufgaben beginnen, die folgenden Teile der Klausur aus:
   a. BEIDE Deckblätter mit Name, Anschrift sowie Matrikelnummer. Markieren Sie vor der Abgabe auf beiden Deckblättern die von Ihnen bearbeiteten Aufgaben.
   b. Falls Sie eine Teilnahmebescheinigung für das Finanzamt wünschen, füllen Sie bitte das entsprechende Formblatt aus.

   Nur wenn Sie beide Deckblätter vollständig ausgefüllt haben, können wir Ihre Klausur korrigieren!
3. Schreiben Sie Ihre Lösungen auf die Aufgabenblätter oder die beigefügten Leerblätter.
4. Als Hilfsmittel sind nur unbeschriebenes Konzeptpapier und Schreibzeug (Füller oder Kugelschreiber) zugelassen.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Aufgabe 1 Unifikation (9 Punkte)

Testen Sie die folgenden Termpaare auf Unifizierbarkeit. Geben Sie, falls vorhanden, den allgemeinsten Unifikator an bzw. begründen Sie, warum kein solcher existiert. (Dabei sind x, y, z Variablen, die anderen Buchstaben sind Konstanten bzw. Funktionssymbole.)

Teil 1 (3 Pkt.):
$f(x, h(y), h(a))$ und $f(g(a), h(g(y)), h(z))$

Teil 2 (3 Pkt.):
$f(x, h(y), h(a))$ und $f(g(a), h(g(b)), h(z))$

Teil 3 (3 Pkt.):
$f(y, h(y), h(a))$ und $f(g(a), h(g(b)), h(z))$

Aufgabe 2 Wahr oder falsch (16 Punkte)

Sind die folgenden Aussagen wahr oder falsch? Begründen Sie Ihre Antwort!

Teil 1 (4 Pkt.):
PROLOG ist eine genaue Umsetzung der Prädikatenlogik **1.** Stufe.

Teil 2 (4 Pkt.):
Funktionale Programme sind immer ineffizienter als entsprechende imperative Programme, weil in die in der funktionalen Programmierung übliche Rekursion mehr Speicher verbraucht: als z.B. eine *for*-Schleife.

Teil 3 (4 Pkt.):
Für die Schemeliste ‘(1 ′a (2 ‘b)) gibt es keine gültige Darstellung als Mirandaliste.

Teil 4 (4 Pkt.):
Miranda ist weniger ausdrucksmächtig als Scheme, weil es in Miranda keine Ströme gibt.

Aufgabe 3 Listenelemente zählen (10 Punkte)

Teil **1** (5 **Pkt.):**

Implernentieren Sie ein PROLOG-Prädikat mit dem Aufrufmuster $ntimes(+E,-N,+L)$, daß genau dann erfüllt ist, wenn N eine natürliche Zahl (einschließlich 0) ist und L eine Liste, die E genau N-mal als Element enthält:

Beispiel:

```
| ?- ntimes(a,N,[]).

N = 0 ? ;

no

| ?- ntimes(a,N,[a,b,c,a,a,g,f,e,a,a]).

N = 5 ? ;

no
```

Teil 2 (5 Pkt.):
Implernentieren Sie eine Scheme-Funktion (ntimes *element liste*), die als Ergebnis die Anzahl der Vorkommen von *element* in der Liste liste zurückliefert.

Beispiel

```
1]=> (ntimes 'a ())

;Value: 0

1]=> (ntimes 'a'(a b c a a g f e a a))

;Value: 5
```

Aufgabe 4 Einlesen von Palindromen (20 Punkte)

Implernentieren Sie ein PROLOG-Prädikat $readpalindrom(-P)$, das solange Listen einliest, bis eine davon ein Palindrom (d.h. identisch zu der umgekehrten Liste) ist. Diese Liste ist beim Verlassen des Prädikats an P gebunden. Durch Backtracking soll kein weiterer Einleseversuch gestartet werden.

Beispiel:

```
| ?- readpalindrom(X).
Palindrom: [a,b].
Kein Palindrom!

Palindrom: [a,b,c].
Kein Palindrom!

Palindrom: [a,b,c,b,a].

X = [a,b,c,b,a] ? ;

no
```

Hinweis: Das Prädikat *reverse* zum Umdrehen von Listen können Sie als gegeben voraussetzen.

Aufgabe 5 Näherungsweise Ableitung (10 Punkte)

Zu einer gegebenen Funktion f kann man die Ableitung an einer Stelle x für ein genügend kleines $k > 0$ näherungsweise wie folgt berechnen:

$$f'(x) \approx \frac{f(x+k) - f(x-k)}{2 \cdot k} =: f'_k(x)$$

Implernentieren Sie eine Scheme-Funktion (ableiten *fkt k),* die eine arithmetische Funktion *fkt* und eine Zahl *k > 0* erwartet und die Funktion f'_k nach obiger Definition zurückliefert.

Beispiel:

```
1 ]=> (define (sqr x) (* x x))

; Value : sqr

1 ]=> (define ablsqr (ableiten sqr 0.1))

; Value : ablsqr

1 ]=> ablsqr

; Value 1:# [compound-procedure 1]

1 ]=> (ablsqr 1)

;Value: 2.0000000000000004

1 ]=> (ablsqr 2)

;Value: 4.000000000000001

1 ]=> (ablsqr 3)

;Value: 6.000000000000005
```

Aufgabe 6 Umgebungen (18 Punkte)

Betrachten Sie folgende Definitionen in der ansonsten leeren Startumgebung eines Scheme-Interpreters:

```
1 ]=> (define x 0)

;Value: x

1 ]=> (define env0
        (make-environment
          (define x 1)
          (define envi
            (make-environment
              (define x 2)))))

;Value: env0
```

Jeder der folgenden Ausdrücke werde direkt nach obigen Definitionen ausgewertet. Geben Sie das Auswertungsergebnis an.

Teil 1 (3 **Pkt.):**
x

Teil 2 (3 Pkt.):
`(eval x env0)`

Teil 3 (3 Pkt.):
`(eval 'x env0)`

Teil 4 (3 Pkt.):
`(eval 'x env1)`

Teil 5 (6 Pkt.):
Geben Sie einen Ausdruck an, der zu dem Wert von x in $env1$ ausgewertet wird.

Aufgabe 7 Constraints (17 Punkte)

In der folgenden Rechnung sind die Ziffern durch Buchstaben ersetzt: wobei unterschiedliche Buchstaben für unterschiedliche Ziffern stehen und keine der dargestellten Zahlen mit einer führenden 0 beginnt.

B	I	E	R	.	S	E	K	T
					T	I	K	R
				K	B	K	R	
			E	1	R	1		
		S	B	R	1			
		S	A	E	U	F	E	R

Implernentieren Sie mittels constraintlogischer Programmierung ein Prädikat $zahlen(?X)$, das genau dann erfüllt ist, wenn X mit einer Liste der Länge 10 instantiiert ist, welche die Ziffern enthält, die den Buchstaben in der Reihenfolge $B, I, E, R, S, K, T, A, U, F$ entsprechen.