zu Aufgabe 1 Unifikation (9 Punkte)

zu Teil 1 (3 Pkt.):
Die beiden Terme $f(x, h(y), h(a))$ und $f(g(a), h(g(y)), h(z))$ sind nicht unifizierbar: Beim Unifikationsversuch der 2. Argumentposition von f sind $h(y)$ und $h(g(y))$ zu unifizieren, was zu einem occur-check-Fehler führt.

zu Teil 2 (3 Pkt.):
Die beiden Terme $f(x, h(y), h(a))$ und $f(g(a), h(g(b)), h(z))$ sind unifizierbar. Der mgu lautet:

$$\sigma_{mgu} = \{x \leftarrow g(a), y \leftarrow g(b), z \leftarrow a\}$$

zu Teil 3 (3 Pkt.):
Die Terme $f(y, h(y), h(a))$ und $f(g(a), h(g(b)), h(z))$ sind nicht unifizierbar: Unifikation der 1. Argumentposition von f führt zur Substitution $\{y \leftarrow g(a)\}$. Damit sind in der 2. Argumentposition $h(g(a))$ und $h(g(b))$ zu unifizieren. Dies führt zum Unifikationsversuch von $g(a)$ und $g(b)$ und im nächsten Schritt von a und b, was einen clash-Fehler liefert.

zu Aufgabe 2 Wahr oder falsch (16 Punkte)

zu Teil 1 (4 Pkt.):
Die Aussage ist falsch: PROLOG enthält u.a. nichtlogische Komponenten wie die fest vorgegebene Suchstrategie, den Cut, Ein-/Ausgabe, Arithmetik und die Datenbankprädikate. All dies ist in der Prädikatenlogik 1. Stufe nicht zu finden.
Andererseits beruht PROLOG auf Horn-Klauseln und nicht auf allgemeinen Klauseln der Prädikatenlogik 1. Stufe. Zum Beispiel ist die Negation der Prädikatenlogik 1. Stufe nicht in Horn-Klausel-Logik und damit auch nicht direkt in PROLOG ausdrückbar.

zu Teil 2 (4 Pkt.):
Die Aussage ist falsch: Eine besondere Klasse von rekursiven Programmen enthält rekursive Aufrufe nur als jeweils *letzten* in einem Berechnungszweig. Derartige Programme lassen sich mit konstantem Speicheraufwand ausführen und damit ebenso effizient wie iterative Programme in imperativen Sprachen.

zu Teil 3 (4 Pkt.):
Die Aussage ist richtig: Das Typsystem von Miranda ist nicht in der Lage, der angegebenen Listenstruktur einen Typ zuzuweisen, da alle Elemente einer Mirandaliste einen gemeinsamen Typ haben müssen.

zu Teil 4 (4 Pkt.):
Die Aussage ist falsch: Da Miranda Terme in Normalordnung auswertet, d.h. die Auswertung von Untertermen aufgeschoben wird, bis sie wirklich gebraucht werden, können (unendliche) Ströme in Miranda als einfache Listen dargestellt werden.

zu Aufgabe 3 Listenelemente zählen (10 Punkte)

zu Teil 1 (5 Pkt.):
Das gesuchte Prädikat definieren wir nach folgender Überlegung:

- Ein beliebiges Element ist in der leeren Liste 0-mal enthalten.
- Ist das 1. Element der Liste gleich E, so muß E im Rest der Liste noch $N-1$-mal vorkommen.
- Ist das 1. Element der Liste nicht E, so muß E noch N-mal im Rest der Liste vorkommen.

Durch Umsetzen dieser 3 Aussagen in PROLOG-Notation erhalten wir:

```
ntimes(_,0,[]).
ntimes(E,N,[E|Rest]) :- ntimes(E,NeuN,Rest), N is NeuN + 1.
ntimes(E1,N,[E2|Rest]) :- E1 \== E2, ntimes(E1,N,Rest).
```

zu Teil 2 (5 Pkt.):
Unsere Scheme-Funktion muß die gleichen drei Fälle betrachten wie das PROLOG-Prädikat in Teil 1:

```
(define (ntimes element liste)
  (cond
    ((null? liste) 0)
    ((eq? element (car liste))
     (1+ (ntimes element (cdr liste))))
    (else
     (ntimes element (cdr liste)))))
```

zu Aufgabe 4 Einlesen von Palindromen (20 Punkte)

```
readpalindrom(P) :- repeat,    % Wiederholen bis erfolgreich
                    write('Palindrom: '), read(P), % Einlesen
                    ispalindrom(P), % Test auf Palindromeigenschaft.
                    !.              % Keine Wiederholung

ispalindrom(P) :- reverse(P,P),!. % Palindrom gefunden
ispalindrom(_) :- write('Kein Palindrom!'), % Falls kein Palindrom,
                  nl, nl,                   % Fehlermeldung
                  fail.                     % und Fehlschlag
```

```
reverse(L,R) :- rev(L,R,[]). % Berechnung mit Akumulator
rev([],R,R).                 % Liste bearbeitet
rev([X|Xs],R,A) :- rev(Xs, R, [X|A]). % rekursiver Fall
```

zu Aufgabe 5 Näherungsweise Ableitung (10 Punkte)

Im wesentlichen besteht die Definition der Funktion *ableiten* aus der Umsetzung der Formel für $f'_k(x)$ in Scheme-Notation. Da das Ergebnis eine Funktion in x sein soll, müssen wir diese Rechenvorschrift anschließend lediglich noch in einem Lambdaausdruck kapseln:

```
(define (ableiten fkt k)
  (lambda (x)
    (/ (- (fkt (+ x k))
          (fkt (- x k)))
       (* 2 k))))
```

zu Aufgabe 6 Umgebungen (18 Punkte)

Wir erhalten folgende Auswertungsergebnisse:

zu Teil 1 (3 Pkt.):

```
1 ]=> x

;Value: 0
```

Zurückgeliefert wird die Bindung von x in der globalen Umgebung.

zu Teil 2 (3 Pkt.):

```
1 ]=> (eval x env0)

;Value: 0
```

Auch hier wird die Bindung von x in der globalen Umgebung zurückgeliefert, da die Auswertung von x vor dem Aufruf von *eval* erfolgt und 0 in der Umgebung *env0* zu 0 ausgewertet wird.

zu Teil 3 (3 Pkt.):

```
1 ]=> (eval 'x env0)

;Value: 1
```

Die Auswertung von x vor dem Aufruf von *eval* wird verzögert. und *eval* wertet x in ***env0*** zu 1 aus.

zu Teil 4 (3 Pkt.):

```
1 ]=> (eval 'x env1)

;Unbound variable: env1
```

Vor dem Aufruf von *eval* wird versucht. *env1* in der globalen Umgebung auszuwerten. wo jedoch keine Bindung verfügbar ist.

zu Teil 5 (6 Pkt.):
Um an die Bindung von x in *env1* zu kommen, muß x in der Umgebung ausgewertet werden, die durch Auswertung von *env1* in *env0* entsteht. der benötigte Ausdruck ist also:

```
1 ]=> (eval 'x (eval 'env1 env0))

;Value: 2
```

zu Aufgabe 7 Constraints (17 Punkte)

Folgendes Programm liefert die Lösung zu unserem Rätsel:

```
zahlen(Buchstaben) :-
      % Liste der Buchstaben generieren
      Buchstaben = [B, I, E, R, S, K, T, A, U, F],

      % weitere Liste für Überträge generieren
      Y = [Ueb1, Ueb2, Ueb3, Ueb4,Ueb5],

      % Domains der Domainvariablen zuweisen.
      domain(Buchstaben,0,9),
      domain(Y,0,4), % Die Addition von 4 Ziffern und 1 Übertrag
                     % liefert Überträge von maximal 4
```

```
% Duplikate in der Belegung der Buchstaben ausschließen
all_different(Buchstaben),


% Führende 0 für alle Zahlen verbieten
S #>= 1, B #>= 1, E #>= 1, K #>= 1, T #>= 1,

% Produkt allgemein berechnen
(1000*B + 100*I +10*E + R) *
(1000*S + 100*E + 10*K + T) #=
(1000000*S + 100000*A + 10000*E + 1000*U + 100*F + 10*E + R),

% Die 4 Zwischenergebnisspalten von oben nach unten.
(1000*B + 100*I +10*E + R) * T #=
(1000*T + 100*I + 10*K + R),

(1000*B + 100*I +10*E + R) * K #=
(1000*K + 100*B + 10*K + R),

(1000*B + 100*I +10*E + R) * E #=
(1000*E + 100*I + 10*R + I),

(1000*B + 100*I +10*E + R) * S #=
(1000*S + 100*B + 10*R + I),

% Die Summation der einzelnen Spalten von hinten nach vorne.
K + R #= E + 10 * Ueb1,
I + K + I + Ueb1 #= F + 10 * Ueb2,
T + B + R + I + Ueb2 #= U + 10 * Ueb3,
K + I + R + Ueb3 #= E + 10 * Ueb4,
E + B + Ueb4 #= A + 10 * Ueb5,
S + Ueb5 #= S,

% Belegung der Constraintvariablen.
labeling([ff],Buchstaben).
```

Die Ausführung dieses Programmes liefert ein eindeutiges Ergebnis:

```
| ?- raetsel(X).

X = [1,0,2,5,6,7,3,4,9,8] ? ;

no
```

D.h. wir erhalten folgende Zuordnung:

$B = 1 \quad I = 0 \quad E = 2 \quad R = 5 \quad S = 6$
$K = 7 \quad T = 3 \quad A = 4 \quad U = 9 \quad F = 8$

und damit

1	0	2	5	·	6	2	7	3
					3	0	7	5
				7	1	7	5	
			2	0	5	0		
		6	1	5	0			
		6	4	2	9	8	2	5