Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Programmiersprachen und -werkzeuge

Sommersemester 2018
18.07.2018

Prof. Dr. Christoph Bockisch
Steffen Dick; Alexander Bille
Johannes Frankenau
Patrick Frömel; Niclas Schmidt
Jonas Stettin; Robert Tran;
Julian Velten

Klausur zur Vorlesung

Deklarative Programmierung

Wichtige Hinweise:

- Schalten Sie, soweit noch nicht geschehen, sofort Ihr Mobiltelefon aus!
- Schalten Sie auch alle anderen medizinisch nicht notwendigen potentiellen Lärmquellen aus.
- Entfernen Sie jetzt alle Gegenstände vom Tisch, außer einen Stift (kein Rot-, Grün-, oder Bleistift), Ihren Studentenausweis und Ihren Personalausweis/Reisepass, Getränke.
- Schreiben Sie jetzt auf jedes Blatt in Druckbuchstaben Ihren Namen und Ihre Matrikelnummer. Blätter ohne Namen ergeben 0 Punkte und werden nicht korrigiert! Füllen Sie insbesondere auch folgende Tabelle in Druckbuchstaben aus:

Vorname	
Nachname	
Matrikelnummer	
Fachbereich	
Studienfach	
Angestrebter Abschluss	

- Im Zweifel sollten Sie Diagramme in der Klausur möglichst detailliert und auf einem möglichst konkreten Niveau erstellen.
- Die Bearbeitungszeit beträgt **2** Stunden.
- Verwenden Sie kein eigenes Papier, für Notizen falls der Platz für die Lösung nicht ausreicht, sondern die Rückseiten. In dringenden Fällen erhalten Sie auf Anfrage Zusatzblätter. Machen Sie gut kenntlich, wenn Sie Rückseiten oder Zusatzblätter für Lösungen verwenden und tragen Sie dort Name und Matrikelnummer ein.
- Es sind keine Hilfsmittel erlaubt. Zuwiderhandlungen führen zum Ausschluss.
- Mehrere widersprüchliche Lösungen zu einer Aufgabe werden mit 0 Punkten bewertet.
- Falls Sie eine Frage haben, so wenden Sie sich bitte leise an einen der Tutoren.
- Die Klausur ist definitiv bestanden mit mindestens 45 Punkten. Mit mindestens 95 Punkten ist definitiv die Note 15 erreicht.

Wird bei der Korrektur ausgefüllt:

Aufgabe	1	2	3	4	5	6	7	8	Total
Punkte	12	13	17	9	17	10	10	12	100
Punkte erreicht									

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 1: Wissensfragen 12 Punkte

Beantworten Sie die folgenden Fragen in 1–2 kurzen Sätzen!

(a) Erklären Sie, was es bei einer **dynamisch getypten Programmiersprache ohne Signaturüberprüfung** schwierig macht, die Ursache von Fehlern zu lokalisieren. **2**

(b) Was ist eine Deduktionsregel und wobei kommt sie zum Einsatz? **2**

(c) Was ist die Kernaussage des DRY Prinzips und was sind die positiven Folgen, wenn es befolgt wird? **2**

(d) Was ist in Prolog eine “existentielle Abfrage”? Beschreiben Sie, was für eine Form und was für eine Bedeutung so eine Abfrage hat. **3**

Vorname:	Nachname:	Matrikelnummer:

(e) Was ist eine Closure? **3**

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 2: Ausdrücke, Werte & Reduktion

13 Punkte

Welches Ergebnis liefern die folgenden Ausdrücke? Bei den Teilaufgaben a) bis c) genügt es, wenn Sie das Ergebnis notieren, die angewendeten Regeln müssen nicht aufgeschrieben werden. Tritt beim Auswerten ein Fehler auf, geben Sie die Ursache des Fehlers an.

(a) `(+ (* (sub1 43) (+ -5 7)) (* 2 3))` **2**

(b)
```
(define x (lambda (x) (* x 2)))
(cond [(< (x (x 2)) 7) 1]
      [(>= (x (x 2)) 7) 2])
```
2

(c) `(string? (number->string (first '("Hello"))))` **2**

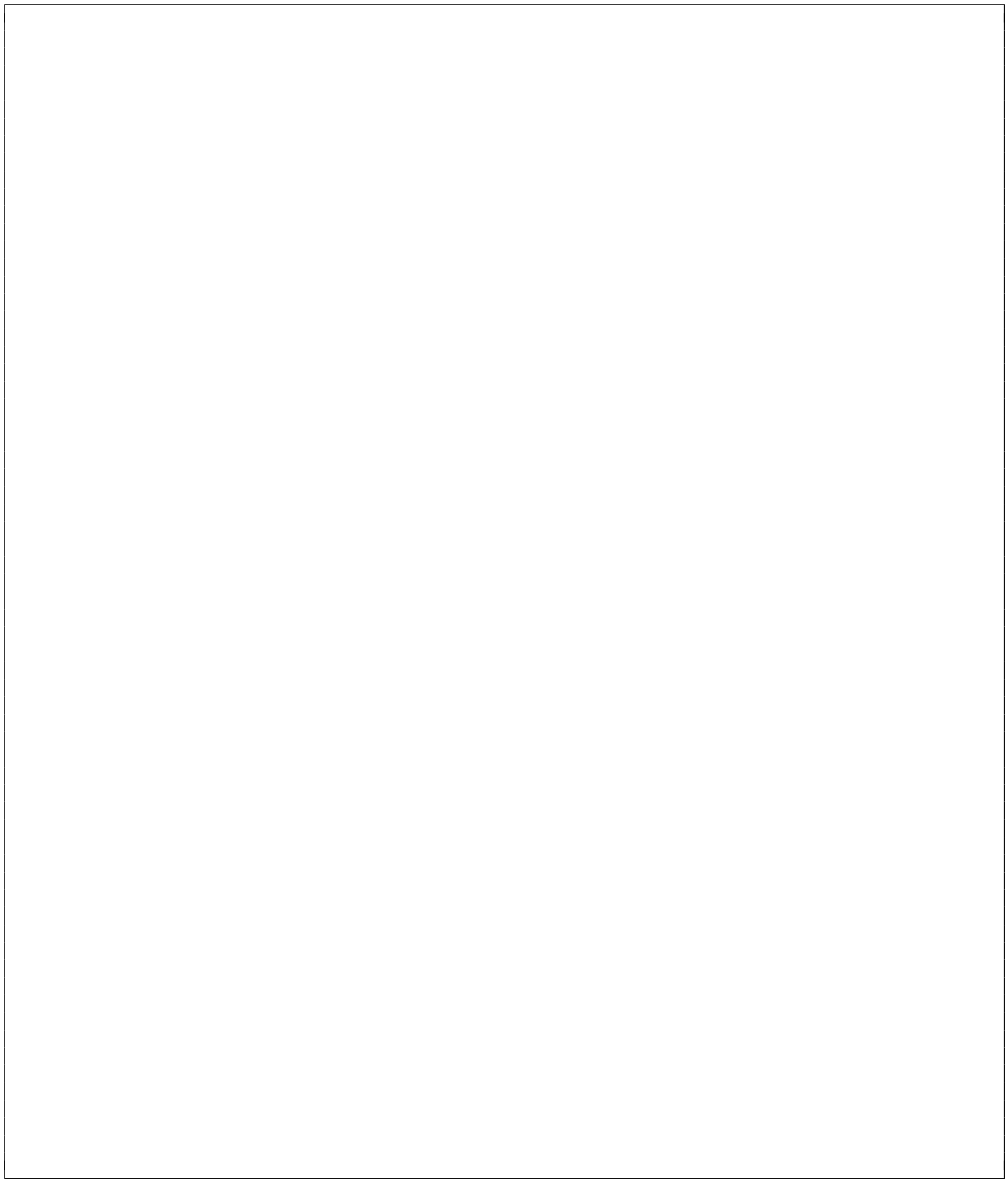
(d) Reduzieren Sie das folgende Programm schrittweise und **unter Angabe der verwendeten Regel**! **7**

```
(define signum (lambda (x)
                 (cond
                   [(= x 0) 0]
                   [(< x 0) -1]
                   [true 1])))
(signum (- 2 4))
```

Hinweis: Notieren Sie dabei in jedem Schritt ausschließlich den relevanten Teil des Programms. Unveränderte Teile können durch „... “ abgekürzt werden.

Hinweis: Sie können für diese Aufgabe den ausgeteilten Zettel mit Auswertungs- und Äquivalenzregeln verwenden.

Vorname:	Nachname:	Matrikelnummer:



Vorname:	Nachname:	Matrikelnummer:

Aufgabe 3: Entwurfsrezept zur Funktionsdef. 17 Punkte

(a) Geben Sie die Signatur der folgenden Funktion an. Verwenden Sie soweit nötig Typparameter, welche die Beziehungen zwischen den Parametern in der Signatur korrekt repräsentieren. Gehen Sie davon aus, dass alle verwendeten Listen homogen sind, also nur Werte von einem Typ enthalten. Begründen Sie alle Teile der Signatur! **8**

```
(define (f x y z)
  (cond [(zero? x) z]
        [else (cons (y x) (f (sub1 x) y z))]))
```

(b) Geben Sie die Signatur (verwenden Sie soweit nötig Typparameter), **Aufgabenbeschreibung** und einen Testfall für die folgende Funktion an. **3**

```
(define (g x m)
  (cond [(> x m) m]
        [true x]))
```

Vorname:	Nachname:	Matrikelnummer:

(c) Gegeben sind folgende Struktur- und Datendefinitionen: **6**

```
(define-struct branch (first second third))
; Ein Branch ist eine Struktur:
; (make-branch Tree Tree Tree)
; interp. ein innerer Knoten in einem ternären Baum.

; Ein Tree ist eins von:
; - Branch
; - Number
; interp. ein ternärer Baum mit Zahlen als Blätter.
```

Geben Sie für eine Funktion `(werteliste t)` Testfälle in ausreichender Anzahl (gemäß der zutreffenden Entwurfsrezepte) und die Implementierung an. Die Funktion, nimmt als Argument einen Tree-Wert und gibt als Ergebnis eine Liste aller in den Blättern des Baumes gespeicherten Werte zurück.

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 4: Algebraische Datentypen 9 Punkte

Entwerfen Sie einen Datentyp zur Speicherung von Adressen. Dabei handelt es sich entweder um deutsche oder amerikanische Anschriften. In beiden Fällen besteht die Adresse aus folgenden Werten: Name, Straße, Hausnummer, Ort. Im Fall der amerikanischen Adressen kommt noch der Bundesstaat hinzu. (Bedenken Sie, dass es auch Hausnummern mit Zusätzen gibt, z.B. “2a”.)

(a) Definieren Sie geeignete Strukturen zur Repräsentation von *AdresseUSA* und *AdresseDeutschland* und geben Sie eine Datendefinition für den Summentyp *Adresse* an. **3**

(b) Definieren Sie den algebraischen Datentyp *Adresse* nochmals mithilfe von `define-type`. **3**

Vorname:	Nachname:	Matrikelnummer:

(c) Implementieren Sie eine Funktion, die eine Adresse in einen formatierten String umwandelt (siehe unten). Sie können annehmen, dass die Konstante `NL` den String für einen Zeilenumbruch enthält. Verwenden Sie zur Fallunterscheidung das `type-case` Konstrukt. **3**

Format deutsche Adressen	**Format USA Adressen**
Name	Name
Straße Hausnummer	Hausnummer Straße
Ort	Ort Bundesstaat

Hinweis: Der Funktion `string-append` kann eine beliebige Anzahl von String-Werten als Argumente übergeben werden, um sie zusammenzusetzen.

```
(define (adresse->string a)
```

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 5: Äquivalenz 17 Punkte

Betrachten Sie die Funktionsdefinitionen in Racket auf der nächsten Seite, welche die Summe aller Quadratzahlen (daher der Name `sq`) bis n berechnen (z.B. $sq(1) = 1$, $sq(2) = 1 + 4 = 5$, etc.). Die Funktion `sq-rec` berechnet das Ergebnis rekursiv, während `sq-form` das Ergebnis über eine Formel ohne Rekursion berechnet.

Es soll die Äquivalenz `(sq-rec n)` $\equiv$ `(sq-form n)` durch strukturelle Induktion über `n` bewiesen werden. Gehen Sie daher davon aus, dass `n` eine natürliche Zahl (inklusive der Null) ist. Geben Sie jeweils die **Namen alle Auswertungs- oder Äquivalenzregeln** an, die sie benutzen.

Hinweis: Die Multiplikation in Racket kann beliebig viele Argumente miteinander multiplizieren. Zum Beispiel entspricht `(* a b c)` der Formel $a \cdot b \cdot c$.

Hinweis: Sie dürfen in Ihrem Beweise die folgende Äquivalenz verwenden:
(EVORGABE) $(n+1) \cdot (n+1) + \frac{n \cdot (n+1) \cdot (2n+1)}{6} \equiv \frac{(n+1) \cdot ((n+1)+1) \cdot (2(n+1)+1)}{6}$

Hinweis: Notieren Sie dabei in jedem Schritt ausschließlich den relevanten Teil des Programms. Unveränderte Teile können durch „... “ abgekürzt werden.

Hinweis: Mehrere arithmetische Umformungen, die direkt hintereinander durchgeführt werden, können Sie zu einem Reduktionsschritt bzw. einem Schritt in der Äquivalenzumformung zusammenfassen.

Hinweis: Sie können für diese Aufgabe den ausgeteilten Zettel mit Auswertungs- und Äquivalenzregeln verwenden.

```
; Number -> Number
(define (sq-rec n)
  (cond [(zero? n) 0]
        [true (+ (* n n)
                 (sq-rec (sub1 n)))]))
```

```
; Number -> Number
(define (sq-form n)
  (/ (* n (+ n 1) (+ (* 2 n) 1)) 6))
```

Vorname:	Nachname:	Matrikelnummer:

(a) Stellen Sie die im Induktionsanfang zu beweisende Äquivalenz auf und führen Sie den Induktionsanfang durch. **7**

(b) Stellen Sie die im Induktionsschritt zu beweisende Äquivalenz und die Induktionsannahme auf. Führen Sie anschließend den Induktionsschritt durch. **10**

Vorname:	Nachname:	Matrikelnummer:

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 6: Listen und Funkt. höherer Ordnung 10 Punkte

Hinweis: Hier finden Sie die Schnittstellenbeschreibung der aus der Vorlesung bekannten Funktionen höherer Ordnung, welche Sie in Aufgabenteil a) verwenden dürfen:

```
; [X] (X -> Boolean) (listof X) -> (listof X)
; Gibt eine Liste zurück, die alle Elemente aus l
; enthält, die das Prädikat p erfüllen
(filter p l)

; [X Y] (X -> Y) (listof X) -> (listof Y)
; Bildet alle Elemente aus l mit f ab und liefert
; die Liste der Ergebnisse.
(map f l)

; [X Y] (X Y -> Y) Y (listof X) -> Y
; Kombiniert alle Elemente der Liste l durch f. Die
; leere Liste wird auf base abgebildet, die Elemente
; werden von rechts nach links durchlaufen.
(foldr f base l)
```

(a) Definieren Sie die nachfolgende **Racket**-Funktion mithilfe einer der oben genannten Funktionen höherer Ordnung. **3**

```
; (listof Number) -> Number
; Berechnet die Anzahl der geraden Zahlen
; in der Liste.
(check-expect (checksum '(42 3 21 10)) 2)
(define (checksum alon)
```

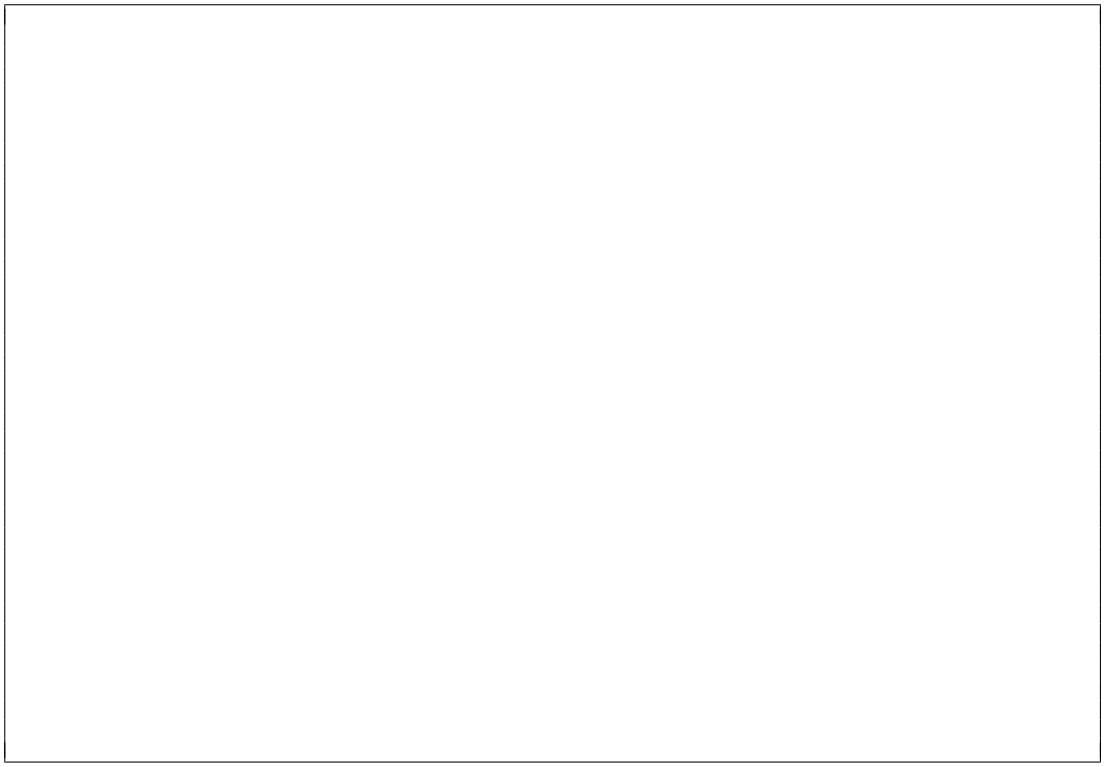
Hinweis: Die Funktion `(even? n)` in Racket gibt `true` zurück, wenn `n` eine gerade Zahl ist.

Vorname:	Nachname:	Matrikelnummer:

(b) Geben Sie eine mögliche Implementierung der Funktion `list-update` (siehe unten) an. Greifen Sie hierbei **nicht** auf vordefinierte Funktionen höherer Ordnung zurück. **7**

```
; [X] (listof X) Number (X -> X)  -> (listof X)
; Gibt eine Liste zurück, die identisch mit der
; übergebenen Liste l ist, bis auf den Eintrag
; an der Position index. Der neue Wert deses
; Elements berechnet sich aus dem ursprünglichen
; Element-Wert durch die in f übergebene Funktion.
; Wenn es kein Element mit dem angegebenen
; index gibt, wird die ; List unverändert
; zurückgegeben.
; Der index des ersten Listen Elements is 0.
(check-expect (list-update '(1 2 3) 1 add1)
     '(1 3 3))
(define (list-update l index f)
```

Vorname:	Nachname:	Matrikelnummer:

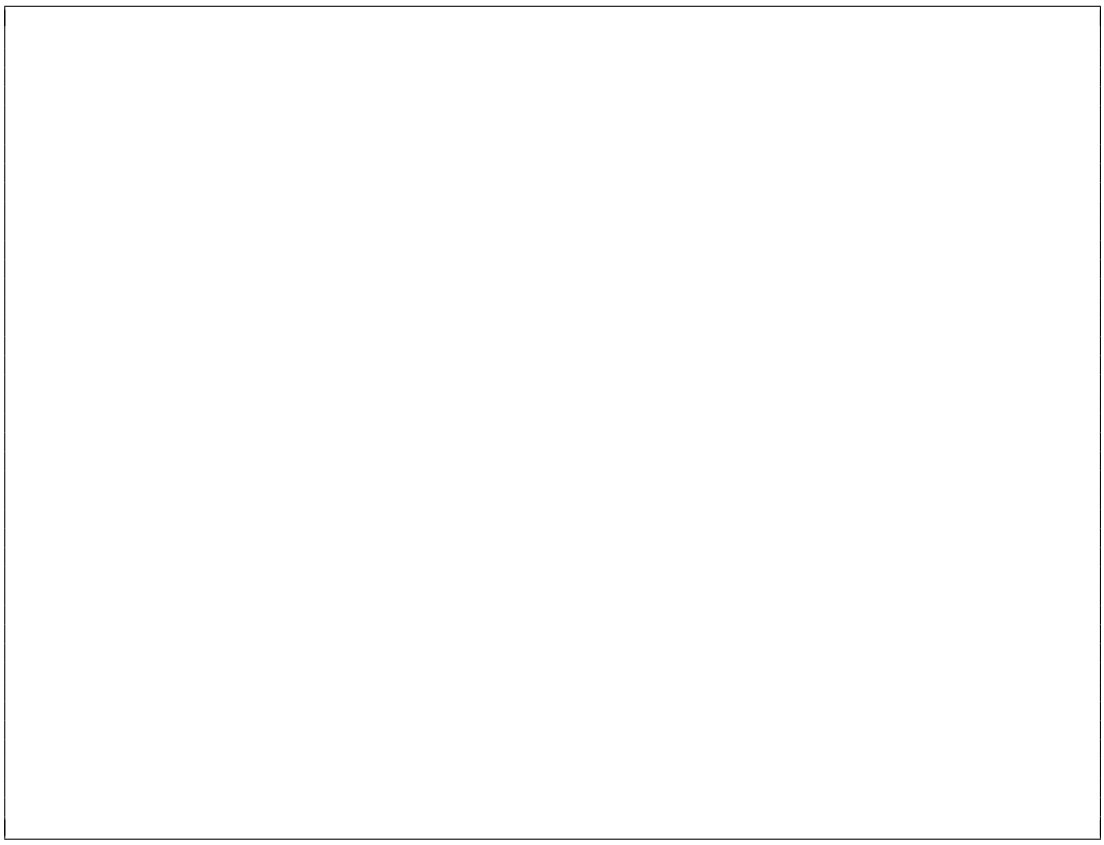


Vorname:	Nachname:	Matrikelnummer:

Aufgabe 7: Rekursion & Terminierung 10 Punkte

(a) Implementieren Sie mittels eines Akkumulators die Funktion `(bucket alon max)`. Diese Funktion gibt eine Liste zurück, die dem Anfang der Liste `alon` entspricht, jedoch ist die Summe der Elemente in der Ergebnisliste kleiner oder gleich `max`. Dabei muss die Ergebnisliste so viele Elemente von `alon` enthalten, wie möglich. **Geben Sie die Akkumulator-Invariante mit an.** **6**

```
; (listof Number) Number -> (listof Number)
(check-expect (bucket '(1 2 3 1 4) 5) '(1 2))
(check-expect (bucket '(1 2 3) 10) '(1 2 3))
(define (bucket alon max)
```



(b) Betrachten Sie die folgende Funktion, welche einen Text erzeugt, welcher mit dem angegebenen Präfix (`prefix`) beginnt gefolgt von zufällig gewählten lateinischen Worten, so dass der Text mindestens die Länge `min-length` hat: **4**

```
(define (random-string prefix min-length)
  (if (>= (string-length prefix) min-length)
      prefix
      (random-string
       (string-append prefix
                      (randomword "latin") " ")
       min-length)))
```

Zum Beispiel könnte der Aufruf `(random-string "" 10)` das Ergebnis `"lorem ipsum "` liefern. Um welche Form der Rekursion handelt es sich bei der Funktion `random-string`? Begründen Sie, warum die Funktion stets terminiert.

Hinweis: Gehen Sie davon aus, dass eine Funktion `(randomword language)` mit der Signatur
`;String -> String` **definiert ist, welche ein zufälliges Wort der angegebenen Sprache zurückliefert.**

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 8: Prolog

12 Punkte

Definieren Sie die folgenden Relation in Prolog mit Hilfe von Rekursion und Pattern-Matching. Die Verwendung von Bibliotheks-Relationen aus Prolog ist dabei **nicht** gestattet. Sollten Sie Hilfsrelationen benötigen, definieren Sie diese selbst.

Hinweis: Die arithmetischen Vergleichsprädikate <, >, =<, >=, =:= (Gleichheit), =\= (Ungleichheit) dürfen verwendet werden und werden in Prolog infix geschrieben. Beispiel: `?- 2 =< 3.`

(a) `merge-zip(L1,L2,L3)` ist erfüllt, wenn die Liste L3 sich folgendermaßen zusammensetzt: Sei n die Länge der kürzeren Liste (L1 oder L2), dann wechseln sich zunächst in L3 die ersten n Elemente aus L1 und L2 ab. Anschließend folgt der Rest der längeren Liste (also die Liste L1 oder L2 ohne die ersten n Elemente). Zum Beispiel ist die Abfrage `?- merge-zip([1,2],[a,b,c,d],[1,a,2,b,c,d]).` erfüllt. **4**

(b) Gegeben seien die folgenden Definitionen der Relationen c1 – c4 und die nachfolgenden Abfragen, die diese verwenden. Geben Sie für jede Abfrage an, was das Resultat ist. Wenn eine Abfrage erfüllt ist, geben Sie eine gültige Substitution aller Variablen an (sowohl der Variablen in der Abfrage als auch in dem entsprechenden Fakt oder der Regel aus dem Prologprogramm). In allen anderen Fällen geben Sie eine kurze Begründung an. **8**

```
b1([Eins,Zwei,Drei]).
b2([A,birne|Rest]).
b3([a,b,X]).
b4(X,Y) :- Y is X + 1.
```

b1) `?- b1([3,2,1]).`

Vorname:	Nachname:	Matrikelnummer:

b2) `?- b2([apfel,birne]).`

b3) `?- b3([a,b,c,d]).`

b4) `?- b4(X,5).`