Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 2 - Lösungsvorschlag 27.10.2008

---

1 Mini Quiz

☒ Elemente einer Liste können von verschiedenen Datentypen sein.

☒ Rekursive Datentypen und Prozeduren brauchen zum Terminieren immer einen Rekursionsanker.

☐ Man kann mit `cons` Listen erzeugen, die man mit `list` nicht erzeugen kann.

☒ Strukturdefinitionen erzeugen automatisch Konstruktoren, Selektoren und Prädikate.

☐ Natürliche Zahlen sind abgeschlossen unter der Subtraktion.

2 Fragen

1. Beschreiben Sie mit eigenen Worten, was ein rekursiver Datentyp ist.
2. Erklären Sie wieso rekursive Datenstrukturen in der Praxis nicht unendlich groß werden.
3. Welche zwei Ansätze beim Design von Programmen mit Hilfe von Wunschlisten sind Ihnen bekannt? Diskutieren Sie Vor- und Nachteile des jeweiligen Ansatzes.
4. Erklären Sie mit eigenen Worten, was Abgeschlossenheit ist.

Lösungsvorschlag:

1. *Eine Rekursion ist der Verweis auf sich selbst. Ein Datentyp ist rekursiv, wenn bei seiner Definition der Datentyp selbst verwendet wird. Zum Beispiel ist ein Strukturtyp, bei dem der Typ eines Elementes wieder der Strukturtyp ist, rekursiv.*
2. *Rekursive Datenstrukturen verwenden stets einen Rekursionsanker, d.h. einen möglichen Wert für ein Element, das selber nicht wieder die Struktur enthält. Bei der Definition der Liste ist dies z.B. die leere Liste, die selber nicht wieder eine Liste enthält. Dies wird durch die Verwendung von heterogenen Daten modelliert. Würde die Definition nicht die leere Liste als Wert für die Struktur enthalten, so ließe sich niemals eine Liste erstellen, da immer weiter Elemente hinzugefügt werden müssten.*

3. *Ausgehend von der Wunschliste aller zu implentierenden (Hilfs-)Prozeduren existieren zwei komplementäre Ansätze fuer die Implementierung des Programmes. Das erste Verfahren ist der sog. Bottom-Up-Ansatz: Es werden zuerst diejenigen Prozeduren implementiert, die von keiner anderen Prozedur auf der Wunschliste abhängen. Der Vorteil besteht darin, dass jede implementierte Prozedur sofort ausgeführt und getestet werden kann. Der Nachteil ist, dass meist nicht klar ist, welche Prozeduren auf der untersten Ebene benötigt werden. Außerdem führt dieser Ansatz oft dazu, dass man sich in Details verliert und nicht mehr das Gesamtziel vor Augen hat.*

   Beim sog. Top-Down-Ansatz wird zuerst die Hauptprozedur der Wunschliste implementiert, und dann die dort aufgerufenen Prozeduren. Der Vorteil bei dieser Methode besteht darin, dass man sich schon zu Beginn auf das Hauptproblem konzentrieren kann, ohne an Details zu denken. Die Details werden dann nach und nach (inkrementell) in Form von Hilfsprozeduren ergänzt. Der Nachteil des Top-Down-Ansatzes ist, dass erst spät getestet werden kann. Man behilft sich oft dadurch, indem man leere Hilfsprozeduren anlegt, die zwar aufgerufen werden können, jedoch nur einen festen Wert zurückliefern. Oft ist jedoch eine Mischung aus beiden Ansätzen sinnvoll
   (Siehe Folien T3.28ff).

4. *Wir sagen: Eine Menge von Elementen ist abgeschlossen bezüglich einer Operation, wenn die Anwendung der Operation auf Elementen der Menge wieder ein Element der Menge produziert. Beispiel: Die Menge der natürlichen Zahlen ist abgeschlossen bezüglich Addition und Multiplikation, da eine Summe oder ein Produkt zweier natürlicher Zahlen wieder eine natürliche Zahl ist. (Siehe Folien T3.10-3.13)*

3 Strukturen (K)

Versuchen Sie bitte die Aufgaben erst ohne einen Rechner zu lösen.
Gegeben sei folgende Strukturdefinition:

```
(define-struct mypair (first second))
```

Werten Sie folgende Ausdrück aus und geben Sie das Ergebnis an.

1. `(make-mypair 'a 'b)`
2. `(mypair? (make-mypair 'a 'b))`
3. `(mypair? (list 'a 'b))`
4. `(make-mypair 1 (make-mypair 2 empty))`
5. `(* (mypair-second (make-mypair 1 2)) (mypair-first (make-mypair 3 4)))`

Lösungsvorschlag:

1. *Konstruktoren sind selbstauswertend* `(make-mypair 'a 'b)`
2. `true`
3. `false`
4. *Konstruktoren sind selbstauswertend* `(make-mypair 1 (make-mypair 2 empty))`
5. `6`

4 Listen (K)

Nehmen Sie für die folgenden Aufgaben das Sprachlevel „Anfänger mit Listen-Abkürzungen“ an. Versuchen Sie bitte die Aufgaben erst ohne einen Rechner zu lösen. Diese Aufgabe muss bearbeitet werden, um die Hausübung lösen zu können.

1. Welche der folgenden Ausdruckspaare werden zu äquivalenten Listen ausgewertet?
   a) `(cons 1 (cons 2 (cons 3 empty)))` und `(list 1 2 3 empty)`
   b) `(cons (list '()) empty)` und `(list 'list empty)`
   c) `(list 7 '* 6 '= 42)` und `(cons 7 (cons '* (cons 6 (cons '= (list 42)))))`
   d) `(cons 'A (list '(I)))` und `(list 'A (cons 'I empty))`

   Lösungsvorschlag:

   a) Die Listen sind nicht äquivalent. Die zweite Liste enthält als viertes Element die leere Liste, die erste Liste enthält nur drei Elemente. Die Auswertung ergibt: `(list 1 2 3)` *und* `(list 1 2 3 empty)`

   b) Die Listen sind nicht äquivalent. Das innere Vorkommen von `list` *wird bei der ersten Liste ausgewertet, bei der zweiten Liste nur als Symbol verwendet. Die Auswertung ergibt:* `(list (list empty))` *und* `(list 'list empty)`

   c) Die Listen sind äquivalent. Die Auswertung ergibt: `(list 7 '* 6 '= 42)`

   d) Die Listen sind äquivalent. Die Auswertung liefert: `(list 'A (list 'I))`

2. Werden folgende Ausdrücke fehlerfrei ausgewertet? Falls nicht, begründen Sie bitte was zum Fehler führt.
   a) `(cons 1 (cons 2 (cons 3)))`
   b) `(cons 1 (list 2 (list '(3 + 4))))`
   c) `(list (cons empty 1) (cons 2 empty) (cons 3 empty))`

   Lösungsvorschlag:

   a) Auswertung nicht möglich, da innerster `cons` *Konstruktor nur einen statt zwei Parametern übergeben bekommt.*

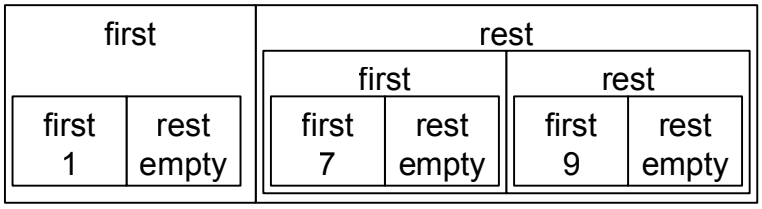
   b) Fehlerfreie Auswertung.

   c) Auswertung nicht möglich, da erster `cons` *Konstruktor als zweiten Parameter eine Liste (und keine Zahl) erwartet.*

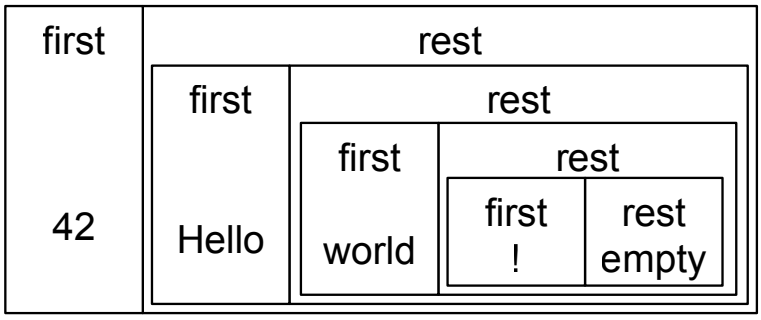
3. In der Vorlesung haben Sie erfahren, wie man Listen in Kästchenschreibweise darstellt (T3.8). Stellen Sie folgende Listen in Kästchenschreibweise dar:
   a) `(define A (list (cons 1 empty) (list 7) 9))`
   b) `(define B (cons 42 (list 'Hello 'world '!)))`

   Lösungsvorschlag:

a)



b)



4. Zu welchen Werten werden folgende Ausdrücke (A und B wie oben definiert) ausgewertet? Falls ein Auswertung nicht möglich ist, begründen Sie bitte.

   a) `(first (rest A))`

   b) `(rest (first A))`

   c) `(first empty)`

   d) `(append (first B) (rest (rest A)) (first A))`

   Lösungsvorschlag:

   a) `(first (rest (list (cons 1 empty) (list 7) 9))) = (first (list (list 7) 9)) = (list 7`

   b) `(rest (first (list (cons 1 empty) (list 7) 9))) = (rest (cons 1 empty)) = empty`

   c) Auswertung nicht möglich, da first eine nicht-leere Liste als Parameter erwartet.

   d) Da `(first B)` *die Zahl 42 zurückliefert und* `append` *nur Listen als Parameter verarbeiten kann, wird die Auswertung mit einem Fehler unterbrochen.*

5. Im folgenden sollen Sie rekursive Prozeduren auf Listen definieren. Vergessen Sie nicht, zuerst Vertrag, Beschreibung und ein Beispiel anzugeben.

   a) Schreiben Sie eine Prozedur `list-length: lst -> num`, die eine Liste als Parameter erhält und die Länge der Liste zurückliefert. (Hinweis: Die leere Liste enthält keine Elemente. Verwenden Sie Rekursion, um das Ergebnis für nicht-leere Listen zu berechnen.)
   Lösungsvorschlag:

```
1  ;; Contract: list-length: lst -> num
2  ;; Purpose:  computes length of a list
3  ;; Example:  (list-length '(a b 0 1)) -> 4
4  (define (list-length lst)
5    (if (empty? lst)
6      ;; list empty? anchor reached,
7      ;; do not recurse further
8      ;; empty list has zero elements so return 0
9      0
10     ;; else, list contains at least one element
```

```
    ;; add one to the length of (rest lst)
    ;; computed by the recursive call
    ;; [(rest lst) is the list without the first element ]
    (+ 1 (list-length (rest lst)))))
;; Test
(list-length '(1 2 3 4 5))
;; should be
5
```

b) Schreiben Sie eine Prozedur `member?: los symbol -> boolean`, die eine Liste von Symbolen und ein Symbol als Parameter erhält und zurückliefert, ob das Symbol in der Liste enthalten ist. (Hinweis: Die leere Liste enthält das Symbol sicher nicht. Verwenden Sie Rekursion, um das Ergebnis für nicht-leere Listen zu berechnen. Verwenden Sie die Prozedur `symbol=?: symbol symbol -> boolean`, um zu überprüfen ob zwei Symbole identisch sind.)
Lösungsvorschlag:

```
;; Contract: member?: los symbol -> boolean
;; Purpose:  computes if symbol is member of los
;; Example:  (member? '(a b) 'b) -> true
(define (member? los symbol)
  (cond
    ;; list empty? anchor reached,
    ;; do not recurse further
    ;; symbol not contained in empty list, return false
    [(empty? los) false]
    ;; else, list contains at least one element
    ;; return true if the first element is equal
    ;; to searched symbol
    [(symbol=? (first los) symbol) true]
    ;; first element is not the searched symbol,
    ;; check if searched symbol is member of
    ;; the rest of list by the recursive call
    [else (member? (rest los) symbol)]))

;; Test
(check-expect (member? '(a b c) 'c) true)
```

c) Schreiben Sie eine Prozedur `remove-duplicates: los -> los`, die eine Liste von Symbolen als Parameter erhält und alle Duplikate aus der Liste entfernt.
Beispiel: `(remove-duplicates '(a a b) -> '(a b)`.
Verwenden Sie Rekursion und betrachten Sie drei Fälle:

- Die leere Liste.
- Das erste Element der Liste kommt im Rest der Liste vor.
- Das erste Element kommt nicht im Rest der Liste vor.

Lösungsvorschlag:

```
;; Contract: remove-duplicates los -> los
;; Purpose:  removes all duplicates from a list of symbols
;; Example:  (remove-duplicates '(a a b)) -> '(a b)
```

```
(define (remove-duplicates list-of-symbols )
  (cond
    ;; list empty? anchor reached,
    ;; do not recurse further
    ;; empty list does not contain duplicates
    ;; and can be returned
    [(empty? list-of-symbols) list-of-symbols]
    ;; first element is a duplicate
    ;; (is a member of the rest of the list)
    [(member? (rest list-of-symbols) (first list-of-symbols))
    ;; so return rest of the list with remaining
    ;; duplicates removed
       (remove-duplicates (rest list-of-symbols))]
    ;; first element is not a duplicate.
    ;; Remove duplicates from the rest of the list
    ;; and add first element at the beginning
    [else (cons
            (first list-of-symbols)
            (remove-duplicates (rest list-of-symbols)))]))

;; Test
(check-expect (remove-duplicates '(a a b)) '(a b))
```

Hausübung

Die Vorlagen für die Bearbeitung werden im GdI1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im GdI1-Portal abgegeben werden. Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe Ihrer Hausübung bestätigen Sie, dass Sie bzw. Ihre Gruppe alleiniger Autor des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.

Abgabe: Spätestens Fr, 07.11.08, 16:00.
Wichtig: Um die volle Punktzahl zu erlangen, müssen Sie jede Ihrer Prozeduren und Hilfsprozeduren mindestens mit Vertrag, Beschreibung und Beispiel kommentieren, sowie jeweils Testfälle angeben. Verwenden Sie als Sprachlevel für die gesamte Hausübung „Anfänger mit Listen-Abkürzungen“.

5 EMail Adresse

1. Tragen Sie eine gültige EMail Adresse im GdI Portal ein („Mein Konto“->Bearbeiten), sonst kann es sein, dass wichtige eMails Ihres Tutors Sie nicht erreichen!!

2. Lesen Sie entweder regelmäßig die EMails in ihrer RBG Mailbox oder richten Sie sich einen Mail forward ein. Sollten Sie Probleme damit haben, folgen Sie den Hinweisen hier: www.rbg.informatik.tu-darmstadt.de/index.php?id=560#forward.

6 Listen und Strukturen für Suchmaschinen (13P)

In dieser Aufgabe werden Teile der Funktionsweise einer Internet-Suchmaschine auf sehr einfache Art und Weise nachgebaut.

Wir beginnen mit der Definition einer Ähnlichkeitsfunktion, der Cosinus-Ähnlichkeit, die wir später benötigen:

```
(define (cosine-similarity vector1 vector2)
  (/
    (vec-mult vector1 vector2)
    (*
      (sqrt (vec-mult vector1 vector1))
      (sqrt (vec-mult vector2 vector2)))))
```

Diese Funktion verwendet die Funktion `vec-mult`, die das Skalarprodukt zweier Vektoren berechnen soll. Vektoren werden als Listen von Zahlen dargestellt. Zur Erinnerung: Das Skalarprodukt zweier Vektoren ist die Summe der Produkte der Komponenten der Vektoren. Also,
`(vec-mult '(1 1 0) '(1 0 1))` = `(+ (* 1 1) (* 1 0) (* 0 1))`.

1. (K) Schreiben Sie die Prozedur `vec-mult`, die zu zwei Vektoren (Listen von Zahlen) das Skalarprodukt berechnet.

   Hinweis: Multiplizieren Sie die jeweils ersten Komponenten der Vektoren und addieren Sie das Ergebnis zum Skalarprodukt der restlichen Vektorkomponenten. Das Skalarprodukt zweier leerer Vektoren ist 0. Sie dürfen voraussetzen, dass die Vektoren gleich viele Komponenten haben.

   - Berechnen Sie `(cosine-similarity '(1 0) '(0 1))`
   - Berechnen Sie `(cosine-similarity '(1 1) '(1 1))`

2. (K) Definieren Sie eine Struktur zum Speichern von Dokumenten. Ein Dokument hat einen Namen und einen Inhalt. Der Inhalt soll hier eine Liste von Symbolen sein (die Wörter des Dokuments, z.B. `'(Dies ist der Inhalt eines Dokuments)`). Definieren Sie zwei Dokumente:

   - `doc1` mit dem Namen `'doc1` und dem Inhalt `'(mouse keyboard screen)`.
   - `doc2` mit dem Namen `'doc2` und dem Inhalt `'(pascal java scheme)`.

3. Schreiben Sie die Funktion `index`, die einen Index zu einer Liste von Dokumenten erzeugt. Ein Index ist eine Liste aller im Inhalt der Dokumente vorkommenden Symbole ohne Duplikate. Verwenden Sie für Dokumente die Struktur aus Aufgabe 6.2. Hinweis: Eine leere Liste von Dokumenten ergibt einen leeren Index. Ist die Liste der Dokumente nicht leer, so kann der Inhalt des ersten Dokumentes mit der Prozedur `append` an den Index der restlichen Dokumente angefügt werden. Zur Erinnerung: Die Prozedur `append` verbindet mehrere Listen, z.B. `(append '(a b) '(c d))` = `'(a b c d)`. Verwenden Sie die Prozedur `remove-duplicates` aus der Präsenzübung um Duplikate aus dem Index zu entfernen.

   - Berechnen Sie `(index (list doc1 doc2))`.
   - Berechnen Sie `(index (list doc1 doc1 doc2))` und überprüfen Sie, dass keine Duplikate vorkommen.
   - Definieren Sie `myindex` als `(index (list doc1 doc2))`.

Der Index aus der vorigen Aufgebe wird verwendet, um Anfragen und Dokumente in Wortvektoren umzuwandeln. Ein Wortvektor zu einem Dokuement oder einer Anfrage enthält für jedes Element des Index eine 0 oder eine 1. Eine 0 bedeutet: Das entsprechende Wort aus dem Index kommt nicht im Dokumentinhalt / der Anfrage vor, eine 1 bedeutet es kommt vor. Dabei spielt es keine Rolle,

dass im Dokumentinhalt / der Anfrage auch Symbole vorkommen, die nicht im Index sind.
Beispiel: Der Index sei `'(this list is an index)` und die Anfrage `'(this is a query)`, so ist der zugehörige Wortvektor `'(1 0 1 0 0)`. Dies bedeutet, dass `'this` und `'is` in der Anfrage vorkommen, alle anderen Symbole aus dem Index nicht.

4. Schreiben Sie eine Funktion `word-vector`, die eine Liste von Symbolen (die Anfrage) oder ein Dokument übergeben bekommt sowie eine zweite Liste von Symbolen (den Index) und den zugehörigen Wortvektor zurückliefert. Hinweis: Implementieren Sie die Prozedur zunächst nur für Anfragen, also Symbollisten, nicht für Dokumente. Verwenden Sie Rekursion. Bei einem leeren Index ist der Wortvektor vollständig und es kann die leere Liste zurückgegeben werden. Bei nicht-leerem Index müssen Sie überprüfen ob das erste Wort im Index in der Symbolliste vorkommt und dementsprechend eine 1 oder 0 vor dem restlichen Wortvektor einfügen. Hierzu können Sie die Prozedur `cons` verwenden. Anschließend fügen Sie zu Beginn der Prozedur eine Behandlung heterogener Daten ein, die im Falle eines Dokumentes als erstem Parameter die Prozedur rekursiv mit dem Dokumenteninhalt aufruft.)
   - Berechnen Sie `(word-vector '(java mouse algol) myindex)`
   - Berechnen Sie `(word-vector doc1 myindex)`

Nun benötigen wir noch eine Struktur zur Ausgabe der Suchergebnisse. Diese soll einen Dokumentnamen zusammen mit einem Punktwert, dem `score` enthalten:

```
(define-struct result (name score))
```

5. Implementieren Sie eine Prozedur `compare`, die eine Suchanfrage (eine Liste von Symbolen), ein Dokument und einen Index erhält und daraus ein `result` erzeugt. Wenden Sie zur Berechnung des `score` die Funktion `cosine-similarity` auf den Wortvektor der Suchanfrage und den Wortvektor des Dokumenteninhaltes an. Für die Erstellung des Wortvektors verwenden Sie den übergebenen Index.

6. Schreiben Sie eine Prozedur `query`, die eine Suchanfrage und eine Liste von Dokumenten übergeben bekommt und eine Liste von Suchergebnissen zurückliefert. Hinweis: Verwenden Sie Rekursion. Sind keine Dokumente in der Liste, so ist das Ergebnis die leere Liste. Ansonsten verwenden Sie die Prozedur `compare`, um das Ergebnis des ersten Dokumentes zu berechnen und `cons` um die restlichen Ergebnisse anzuhängen.