Grundlagen der Informatik 1
WS 2008/09

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

Übung 3 Lösungsvorschlag v1.1 03.11.2008

Änderungen

- 08.11.2008: Aufgabe 5.1.3 Wurzel eines binären Codierungsbaums als zweiten Parameter für `encode-list` hinzugefügt.

1 Mini Quiz

Kreuzen Sie die wahren Aussagen an!

☒ Strukturdefinitionen können mit `local` innerhalb von Funktionen erfolgen.

☒ Eine Baumstruktur in Scheme kann verschiedene Datentypen in ihren Knoten speichern.

☐ Innerhalb eines `local` Ausdrucks kann nicht auf Definitionen außerhalb des `local` Ausdruck zugegriffen werden.

☒ Der Scope einer Namensbindung ist der textuelle Bereich, in dem sich ein Auftreten des Namens auf diese Namensbindung bezieht.

2 Fragen

1. Welche Richtlinien sollten bei der Benutzung von `local` beachtet werden?
2. Was macht aus Software-Engineering Sicht hochwertigen Code aus?
3. Erläutere Probleme, die entstehen können, wenn Daten redundant abgelegt werden.

Lösungsvorschlag:

1. *T4.29: Wenn Sie bei der Entwicklung einer Prozedur nach dem Designrezept feststellen, dass Sie Hilfsprozeduren benötigen, die sie lediglich innerhalb der zu schreibenden Prozedur benötigen, definieren Sie diese auch nur innerhalb eines local Blocks.*

   Nutzen Sie aus, dass Sie innerhalb des local Blocks Zugriff auf lexikalisch weiter außen liegende Namen (z.B. Prozedurparameter) haben.

   Wenn Sie innerhalb eines Prozedurkörpers einen Wert mehrfach ausrechnen, definieren Sie einen lokalen Namen mit diesem Wert und benutzen ihn an allen Stellen, wo sie diesen Wert zuvor berechneten.

2. *aus Software-Engineering Sicht hochwertiger Code ist gut erweiterbar. Es sollte einfach sein, neue Datentypen und neue Operationen hinzuzufügen. Einfach ist das immer dann, wenn möglichst wenig bestehender Code geändert werden muss.*

3. *Werden Daten redundant abgelegt, so muss bei einer Code-Erweiterung, die diese Daten betrifft an allen Stellen etwas geändert werden. Dies führt zu aus Software-Engineering Sicht schlechtem Code.*

3 λocalverbot

Hintergrund: RGB Der RGB Code eines Farbton ist ein Tripel $(R, G, B)R, G, B \in [0; 255]$. Die Werte R, G und B sind dabei die Anteile der Grundfarben Rot, Grün und Blau am Farbton. Sie reichen von 0 (diese Grundfarbe ist gar nicht im Farbton vorhanden) bis 255 (diese Farbe ist in voller Intensität im Farbton vorhanden). Beispiel: (255 0 0) reines Rot, (114 247 160) Hellgrün. Der RGB Code eignet sich gut, um Farben für die Darstellung auf einem Bildschirm zu speichern. Der vom Menschen empfundene Farbton lässt sich aus einem RGB Tripel nur schwer ersehen. Was für eine Farbe istz.B. (204, 102, 153)?. Um dieses Problem zu lösen, kann man den Winkel H (H für hue/Farbe) zu einem RGB Tripel berechnen. Ein H in der Nähe von 0° bedeutet z.B. rötlich, ein H in der Nähe von 240° bedeutet bläulich. Aus Wikipedia stammen folgende Formeln zur Errechnung des Winkels H zu einem RGB Tripel (R, G, B):

$$r = R/255, g = G/255, b = B/255$$

$$M = max(r, g, b), m = min(r, g, b)$$

$$H = \begin{cases} 0° & \text{if } M = m \ , \\ 60°(g-b)/(M-m) & \text{if } r = M \ , \\ 120° + 60°(b-r)/(M-m) & \text{if } g = M \ , \\ 240° + 60°(r-g)/(M-m) & \text{if } b = M \end{cases}$$

Lösungsvorschlag:
(204, 102, 153) ist pink
Scheme verwendet die Struktur `color` zum speichern von RGB Tripeln. Folgende Scheme Prozedur berechnet H.

```
1  ;; Diese Funktion berechnet aus einer RBG Farbe den hue Wert
2  ;; http://de.wikipedia.org/wiki/HSI-Farbmodell
3  ;; http://de.wikipedia.org/wiki/RGB-Farbraum
4  (define (hue color)
5    (local (
6      (define r (/ (color-red color) 255))
7      (define b (/ (color-blue color) 255))
8      (define g (/ (color-green color) 255))
9      (define (h x)
10       (if (< x 0) (+ x 360) x)))
11     (local (
12         (define MAXIMUM (max r b g))
```

```
13          (define MINIMUM (min r b g)))
14          (local (
15             (define (f a b)
16               (* (/ (- a b) (- MAXIMUM MINIMUM)) 60)))
17               (cond
18                   [(= MINIMUM MAXIMUM) 0]
19                   [(= r MAXIMUM) (h (f g b)) ]
20                   [(= g MAXIMUM) (h (+ (f b r) 120))]
21                   [else (h (+ (f r g) 240)) ])))))
```

Auftrag Der Software Entwickler „Schematics“ gibt Ihnen den Auftrag die Prozedur `hue` so umzuändern, so dass sie ohne die Verwendung von `local` Ausdrücken auskommt, da diese der Firmenpolitik widersprechen. Hilfsprozeduren sollen auch nicht verwendet werden, da sie nur unnötigen Dokumentationsaufwand verursachen. Ersetzen Sie eine `local` Definition nach der anderen. Was halten Sie von der Firmenpolitik von „Schematics“?

Lösungsvorschlag:

```
1  ;; Diese Funktion wandelt eine RBG Farbe in HSL um
2  ;; http://de.wikipedia.org/wiki/HSI-Farbmodell
3  ;; http://de.wikipedia.org/wiki/RGB-Farbraum
4  (define (hue_no_local color)
5        (cond
6          [(= (max (/ (color-red color) (+ (color-red color) (color-green
             color) (color-blue color))) (/ (color-green color) (+ (color-red
              color) (color-green color) (color-blue color))) (/ (color-blue
             color) (+ (color-red color) (color-green color) (color-blue
             color)))) (min (/ (color-red color) (+ (color-red color) (color-
             green color) (color-blue color))) (/ (color-green color) (+ (
             color-red color) (color-green color) (color-blue color))) (/ (
             color-blue color) (+ (color-red color) (color-green color) (
             color-blue color))))) 0]
7
8          [(= (/ (color-red color) (+ (color-red color) (color-green color)
              (color-blue color))) (max (/ (color-red color) (+ (color-red
             color) (color-green color) (color-blue color))) (/ (color-green
             color) (+ (color-red color) (color-green color) (color-blue
             color))) (/ (color-blue color) (+ (color-red color) (color-green
              color) (color-blue color))))    (* (/ (- (/ (color-green color
             ) (+ (color-red color) (color-green color) (color-blue color)))
             (/ (color-blue color) (+ (color-red color) (color-green color) (
             color-blue color)))) (- (max (/ (color-red color) (+ (color-red
             color) (color-green color) (color-blue color))) (/ (color-green
             color) (+ (color-red color) (color-green color) (color-blue
             color))) (/ (color-blue color) (+ (color-red color) (color-green
              color) (color-blue color)))) (min (/ (color-red color) (+ (
             color-red color) (color-green color) (color-blue color))) (/ (
             color-green color) (+ (color-red color) (color-green color) (
             color-blue color))) (/ (color-blue color) (+ (color-red color) (
             color-green color) (color-blue color)))))) 60)]
9
10         [(= (/ (color-green color) (+ (color-red color) (color-green
             color) (color-blue color))) (max (/ (color-red color) (+ (color-
             red color) (color-green color) (color-blue color))) (/ (color-
             green color) (+ (color-red color) (color-green color) (color-
```

```
        blue color))) (/ (color-blue color) (+ (color-red color) (color-
        green color) (color-blue color))))) (+ (* (/ (- (/ (color-blue
        color) (+ (color-red color) (color-green color) (color-blue
        color))) (/ (color-red color) (+ (color-red color) (color-green
        color) (color-blue color)))) (- (max (/ (color-red color) (+ (
        color-red color) (color-green color) (color-blue color))) (/ (
        color-green color) (+ (color-red color) (color-green color) (
        color-blue color))) (/ (color-blue color) (+ (color-red color) (
        color-green color) (color-blue color)))) (min (/ (color-red
        color) (+ (color-red color) (color-green color) (color-blue
        color))) (/ (color-green color) (+ (color-red color) (color-
        green color) (color-blue color))) (/ (color-blue color) (+ (
        color-red color) (color-green color) (color-blue color)))))) 60)
         120)]

      [else    (+ (* (/ (- (/ (color-red color) (+ (color-red color) (
        color-green color) (color-blue color))) (/ (color-green color)
        (+ (color-red color) (color-green color) (color-blue color))))
        (- (max (/ (color-red color) (+ (color-red color) (color-green
        color) (color-blue color))) (/ (color-green color) (+ (color-red
         color) (color-green color) (color-blue color))) (/ (color-blue
        color) (+ (color-red color) (color-green color) (color-blue
        color)))) (min (/ (color-red color) (+ (color-red color) (color-
        green color) (color-blue color))) (/ (color-green color) (+ (
        color-red color) (color-green color) (color-blue color))) (/ (
        color-blue color) (+ (color-red color) (color-green color) (
        color-blue color)))))) 60) 240)]))
```

4 Tree Sort

Diese Aufgabe hilft bei der Lösung der Hausübung!
In dieser Aufgabe werden Sie den TreeSort Algorithmus zum sortieren von Listen implementieren. Dazu wird eine in der Informatik sehr oft verwendete Datenstruktur, der Binärbaum benötigt.

Hintergrund: Die Datenstruktur sortierter Binärbaum

Ein Baum ist eine rekursive Datenstruktur, die folgendermaßen definiert ist: Jeder *Baumknoten* enthält einen Inhalt, einen linken Sohn und einen rechten Sohn. Linker und rechter Sohn sind dabei auch wieder *Baumknoten*. Als Rekursionsanker dient der „leere Baum“, `empty`, der per Definition ein Baumknoten ist. Sind beide Söhne eines Baumknotens `empty`, so nennt man diesen Baumknoten ein „Blatt“. Der oberste Baumknoten, der selber nicht Sohn eines weiteren Baumknoten ist heißt „Wurzel“ des Baumes.
Zusätzlich gilt bei einem sortierten binären Baum folgende Bedingung: Sei n ein Bauknoten. Der Inhalt des linken Sohns von n ist ein sortierter Binärbaum, dessen größtes Blatt stets kleiner oder gleich dem Inhalt von n oder der linke Sohn ist der „leere Baum“; der Inhalt des rechten Sohns von n ist ein sortierter Binärbaum, dessen kleinstes Blatt stets größer als der Inhalt von n oder der rechte Sohn von n ist der „leere Baum“. Vergleichen Sie dazu folgende Scheme Definition und die Beispiele:

```
1 ;; struct for storing binary trees
2 (define-struct treenode (left content right))
3
4 ;;example binary tree with three nodes
```

```
;;     2
;;    / \
;;   1   3
;; / \ / \
;;
(make-treenode (make-treenode empty 1 empty) 2 (make-treenode empty 3
  empty))

;;not a sorted binary tree! Left son is larger than the node content!
;;     2
;;    / \
;;   4   3
;; / \ / \
;;
(make-treenode (make-treenode empty 4 empty) 2 (make-treenode empty 3
  empty))
```

4.1 Sortieren von Zahlen

Lesen Sie zunächst aufmerksam alle Teilaufgaben durch bevor sie mit der Implementierung der Lösung beginnen. Entscheiden Sie sich für ein Vorgehen top-down oder bottom-up und machen sich eine Wunschliste von Funktionen. Verwenden Sie `local` für Funktionen die nicht nach außen sichtbar sein sollen.

1. Implementieren Sie eine Funktion `sort-list`, die alle Zahlen in einer Liste zuerst in einen sortierten Binärbaum einfügt und dann als sortierte Liste wieder zurück gibt. **Lösungsvorschlag:**

```
;;contract: lon -> lon
;;purpose:  sorts a list of numbers
;;example: (sort-list '(1 3 2)) -> '(1 2 3)
(define (sort-list L)
  (local (
          (define-struct treenode (left content right))
          ;;contract: treenode list -> treenode
          ;;purpose:  inserts all elements of list into the sorted
            binary tree root and returns the new sorted binary tree
          ;;example: (tree-insert-list empty '(3)) -> (make-treenode
            empty 3 empty)
          (define (tree-insert-list root L)
            (local (
                    ;;contract: treenode num -> treenode
                    ;;purpose:  inserts content into the sorted
                      binary tree root and returns the new sorted
                      binary tree
                    ;;example: (tree-insert empty 3) -> (make-
                      treenode empty 3 empty)
                    (define (tree-insert root content)
                      (cond
                        [(empty? root) (make-treenode empty content
                          empty)]
                        [(<= content (treenode-content root)) (make-
                          treenode (tree-insert (treenode-left root)
                          content) (treenode-content root) (treenode-
                          right root))]
```

```
                        [else (make-treenode (treenode-left root) (
                          treenode-content root) (tree-insert (
                          treenode-right root) content))])))
            (if (empty? L) root (tree-insert-list (tree-insert root
              (first L)) (rest L)))))
        ;;contract: tree -> list
        ;;purpose:  outputs all elements of the tree in infix order
        ;;example: (flatten-tree empty (tree-insert-list empty '(5
          6))) -> '(5 6)
        (define (flatten-tree root)
          (if (empty? root)
              empty
              (append (flatten-tree (treenode-left root)) (cons (
                treenode-content root) (flatten-tree (treenode-
                right root)))))))
  (flatten-tree (tree-insert-list empty L))))

;;test
(check-expect (sort-list '(5 6)) '(5 6))
;;test
(check-expect (sort-list '(6 5)) '(5 6))
```

2. Implementieren Sie eine Funktion `tree-insert-list`, die eine Liste von Zahlen in einen sortierten Binärbaum einfügt.

Lösungsvorschlag:

```
(define-struct treenode (left content right))

;;contract: treenode list -> treenode
;;purpose:  inserts all elements of list into the sorted binary tree
  root and returns the new sorted binary tree
;;example: (tree-insert-list empty '(3)) -> (make-treenode empty 3
  empty)
(define (tree-insert-list root L)
  (local (
          ;;contract: treenode num -> treenode
          ;;purpose:  inserts content into the sorted binary tree
            root and returns the new sorted binary tree
          ;;example: (tree-insert empty 3) -> (make-treenode empty 3
            empty)
          (define (tree-insert root content)
            (cond
              [(empty? root) (make-treenode empty content empty)]
              [(<= content (treenode-content root)) (make-treenode (
                tree-insert (treenode-left root) content) (treenode-
                content root) (treenode-right root))]
              [else (make-treenode (treenode-left root) (treenode-
                content root) (tree-insert (treenode-right root)
                content))])))
    (if (empty? L) root (tree-insert-list (tree-insert root (first L)
      ) (rest L)))))

;;test
(check-expect
   (tree-insert-list empty '(5))
```

```
21    (make-treenode empty 5 empty))
22
23  ;;test
24  (check-expect
25    (tree-insert-list empty '(5 6))
26    (make-treenode empty 5 (make-treenode empty 6 empty)))
```

3. (K) Implementieren Sie eine Funktion `tree-insert`, die die Wurzel $root$ eines sortierten Binärbaums T und eine Zahl n übergeben bekommt und die die Wurzel eines neuen sortierten Binärbaum T' zurück liefert, der alle Elemente von T sowie n enthält. Beachten Sie dabei die Regeln für sortierte binäre Bäume, nachdem der linke Sohn immer kleiner und der rechte Sohn immer größer als der Inhalt des Vaters sein muss!

Lösungsvorschlag:

```
1  (define-struct treenode (left content right))
2
3  ;;contract: treenode num -> treenode
4  ;;purpose:  inserts content into the sorted binary tree root and
     returns the new sorted binary tree
5  ;;example: (tree-insert empty 3) -> (make-treenode empty 3 empty)
6  (define (tree-insert root content)
7    (cond
8      [(empty? root) (make-treenode empty content empty)]
9      [(<= content (treenode-content root)) (make-treenode (tree-insert
         (treenode-left root) content) (treenode-content root) (
       treenode-right root))]
10     [else (make-treenode (treenode-left root) (treenode-content root)
         (tree-insert (treenode-right root) content))]))
11
12 ;;Test
13 (check-expect (tree-insert empty 5) (make-treenode empty 5 empty))
14
15 ;;Test
16 (check-expect (tree-insert (make-treenode empty 5 empty) 6) (make-
     treenode empty 5 (make-treenode empty 6 empty)))
```

4.2 Sortieren von Studenten

Implementieren Sie nun Varianten von `sort-list`, um damit Studenten nach verschiedenen Kriterien sortieren zu können.

1. Überlegen Sie sich vorab eine Struktur zur Speicherung von Studenten mit Namen und Geburtsdatum.

2. Erstellen sie zwei Varianten von `sort-list`, die Listen von Studenten sortieren.
   a) Sortieren Sie die Studenten nach ihren Matrikelnummern
   b) Sortieren Sie die Studenten nach ihrem Alter

Lösungsvorschlag:

```
1  ;; struct for storing student records
2  (define-struct student (name matrikel birthyear birthmonth birthday))
3
4  ;; some example students
5  (define melanie (make-student 'melanie 1234567 1968 6 12))
6  (define daniel (make-student 'daniel 2234567 1982 8 5))
7  (define guido (make-student 'guido 3234567 1980 5 5))
8  (define gina (make-student 'gina 4234567 1980 5 7))
9
10
11 ;;----------------------------------------
12 ;;sort students by matrikel
13 ;;----------------------------------------
14
15 ;;contract: list-of-students -> list-of-students
16 ;;purpose:  sorts a list of students by matrikel
17 (define (sort-student-list-matrikel L)
18   (local (
19           (define-struct treenode (left content right))
20           ;;contract: treenode list -> treenode
21           ;;purpose:  inserts all elements of list into the sorted
               binary tree root and returns the new sorted binary tree
22           (define (tree-insert-list root L)
23             (local (
24                     ;;contract: student student -> boolean
25                     ;;purpose:  return true if the first student's
                         matrikel is less or equal than the second
                         student's matrikel
26                     ;;example: (comp-student-matrikel (make-student '
                         melanie 1234567 1968 6 12)) (make-student '
                         daniel 2234567 1982 8 5))) -> true
27                     (define (comp-student-matrikel stud1 stud2)
28                       (<= (student-matrikel stud1) (student-matrikel
                           stud2)))
29                     ;;contract: treenode num -> treenode
30                     ;;purpose:  inserts content into the sorted
                         binary tree root and returns the new sorted
                         binary tree
31                     (define (tree-insert root content)
32                       (cond
33                         [(empty? root) (make-treenode empty content
                             empty)]
34                         [(comp-student-matrikel content (treenode-
                             content root)) (make-treenode (tree-insert
                             (treenode-left root) content) (treenode-
                             content root) (treenode-right root))]
35                         [else (make-treenode (treenode-left root) (
                             treenode-content root) (tree-insert (
                             treenode-right root) content))])))
36               (if (empty? L) root (tree-insert-list (tree-insert root
                     (first L)) (rest L)))))
37           ;;contract: tree -> list
38           ;;purpose:  outputs all elements of the tree in infix order
39           ;;example: (flatten-tree empty (tree-insert-list empty '(5
               6))) -> '(5 6)
40           (define (flatten-tree root)
41             (if (empty? root)
```

```
42                  empty
43                  (append (flatten-tree (treenode-left root)) (cons (
                      treenode-content root) (flatten-tree (treenode-
                      right root)))))))
44    (flatten-tree (tree-insert-list empty L))))
45
46  ;; Test
47  (check-expect
48   (sort-student-list-matrikel (list daniel gina guido melanie))
49   (list melanie daniel guido gina))
50
51
52  ;;-----------------------------------------
53   ;; sort students by age
54  ;;-----------------------------------------
55
56   ;; contract: list-of-students -> list-of-students
57   ;; purpose:  sorts a list of students by matrikel
58   (define (sort-student-list-age L)
59     (local (
60              (define-struct treenode (left content right))
61              ;; contract: treenode list -> treenode
62              ;; purpose:  inserts all elements of list into the sorted
                  binary tree root and returns the new sorted binary tree
63              (define (tree-insert-list root L)
64                (local (
65                         ;; contract: student student -> boolean
66                         ;; purpose:  return true if the first student's
                            age is less or equal than the second student's
                            age
67                         ;; example: (comp-student-age (make-student '
                            melanie 1234567 1968 6 12)) (make-student '
                            daniel 2234567 1982 8 5))) -> false
68                        (define (comp-student-age stud1 stud2)
69                          (or
70                              (< (student-birthyear stud1) (student-
                                  birthyear stud2))
71                              (and
72                                (= (student-birthyear stud1) (student-
                                    birthyear stud2))
73                                (< (student-birthmonth stud1) (student-
                                    birthmonth stud2)))
74                              (and
75                                (= (student-birthyear stud1) (student-
                                    birthyear stud2))
76                                (= (student-birthmonth stud1) (student-
                                    birthmonth stud2))
77                                (< (student-birthday stud1) (student-
                                    birthday stud2)))))
78                        ;; contract: treenode num -> treenode
79                        ;; purpose:  inserts content into the sorted
                            binary tree root and returns the new sorted
                            binary tree
80                        (define (tree-insert root content)
81                          (cond
82                            [(empty? root) (make-treenode empty content
                                empty)]
83                            [(comp-student-age content (treenode-content
```

```
                                root)) (make-treenode (tree-insert (
                                treenode-left root) content) (treenode-
                                content root) (treenode-right root))]
84                            [else (make-treenode (treenode-left root) (
                                treenode-content root) (tree-insert (
                                treenode-right root) content))])))
85                  (if (empty? L) root (tree-insert-list (tree-insert root
                      (first L)) (rest L)))))
86              ;;contract: tree -> list
87              ;;purpose:  outputs all elements of the tree in infix order
88              ;;example: (flatten-tree empty (tree-insert-list empty '(5
                  6))) -> '(5 6)
89              (define (flatten-tree root)
90                (if (empty? root)
91                    empty
92                    (append (flatten-tree (treenode-left root)) (cons (
                      treenode-content root) (flatten-tree (treenode-
                      right root)))))))
93    (flatten-tree (tree-insert-list empty L))))
94
95  ;; Test
96  (check-expect
97   (sort-student-list-matrikel (list daniel gina guido melanie))
98   (list melanie daniel guido gina ))
```

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.

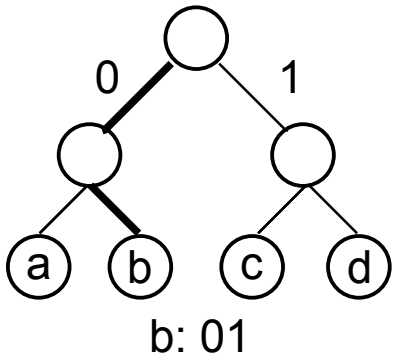
Abgabe: Spätestens Fr, 14.11.08, 16:00. Wichtig: Um die volle Punktzahl zu erlangen, müssen Sie jede Ihrer Prozeduren und Hilfsprozeduren mindestens mit Vertrag, Beschreibung und Beispiel kommentieren, sowie jeweils Testfälle angeben. Verwenden Sie als Sprachlevel für die gesamte Hausübung „Zwischenstufe“.

5 Datenkompression

Codierung

Codierung ist die Darstellung von Symbolen in einem Alphabet durch Folgen von Nullen und Einsen. Eine bekannte Codierung ist z.B. ASCII. Im ASCII Code werden alle Symbole durch eine Folge von acht Nullen und Einsen codiert. Ein anderer bekannter Code ist der Morse-Code. Hier wird das e mit der kurzen Folge 0 (. = 0) codiert, das seltene q hingegen mit 1101 (- - . -; 1 = -). Werden seltene Symbole durch lange Codes und häufige Symbole durch kurze Codes dargestellt, sinkt die Gesamtlänge der Codesequenz für Nachrichten die aus vielen Symbolen bestehen. Diese Methode wird in Datenkompressionsverfahren, wie z.B. dem ZIP Algorithmus verwendet.
Als Basis der Codierung dienen binäre Codierungsbäume. In den Blättern dieser Bäume stehen die zu codierenden Symbole. Möchte man den Code zu einem Symbol ermitteln, so beginnt man bei

der Wurzel des Baumes und durchläuft den Baum bis zum Blatt in dem das Symbol steht. Geht man dabei nach links, so fügt man eine „0“ an den Code an, geht man nach rechts, so fügt man eine „1“ an.



5.1 Encode (5P)(K)

In der Vorlage ist ein einfacher Codierungsbaum `simplecode` enthalten. Benutzen Sie diesen zum Testen ihrer Prozeduren.

1. Schreiben Sie eine Funktion `contains?`, die überprüft, ob ein Symbol als Blatt in einem binären Codierungsbaum vorkommt. Verwenden Sie Rekursion über die Baumstruktur: Ein Symbol kommt in einem binären Codierungsbaum vor, wenn es im Wurzelknoten steht oder im linken oder rechten Sohn vorkommt.

2. Schreiben Sie eine Funktion `encode`, die ein Symbol und die Wurzel eines binären Codierungsbaums übergeben bekommt. Als Ergebnis soll die Codierung des Symbols mit dem Codierungsbaum zurück geliefert werden. Verwenden Sie Rekursion über die Baumstruktur, d.h. wenn das Symbol in der Wurzel vorkommt wird der bisher erzeugte Code zurück geliefert. Kommt das Symbol im linken Teilbaum vor, so fügen Sie eine „0“ in den Code ein und rufen `encode` rekursiv mit dem linken Sohn auf. Kommt das Symbol im rechten Teilbaum vor, so fügen Sie eine „1“ in den Code ein und rufen `encode` rekursiv mit dem rechten Sohn auf.

3. Schreiben Sie eine Funktion `encode-list`, die eine Liste von Symbolen und die Wurzel eines binären Codierungsbaums übergeben bekommt und eine Liste mit den Codes der Symbole zurück liefert.

5.2 Komprimierende Codes (8P)

Der Codierungsbaum `simplecode` erzeugt zu allen Symbolen gleichlange Codes. Um Daten zu komprimieren sollte man aber, wie beim Morse Code, für häufig vorkommende Symbole kurze Codes wählen und dafür für seltene Symbole längere Codes erlauben. Um die Codelänge optimal auf die Symbolhäufigkeit abzustimmen, werden Wahrscheinlichkeitsbäume als Codebäume verwendet. In einem binären Wahrscheinlichkeitsbaum enthält der Inhalt jedes Knotens eine Wahrscheinlichkeit, die gleich der Summe der Wahrscheinlichkeiten des linken und rechten Sohnes ist. Die Struktur `ptree-nodes` aus der Vorlage soll für den Knoteninhalt von Wahrscheinlichkeitsbäumen verwendet werden.
Die Liste `freq` von `ptree-nodes` aus der Vorlage enthält die Wahrscheinlichkeit des Auftretens von Buchstaben in einem englischen Text. Die Vorlage enthält ausserdem die Funktion `make-subtree`,

die aus einem `ptree-node` einen einelementigen Teilbaum erzeugt. Bekommt `make-subtree` einen Baum übergeben, so bleibt dieser unverändert.

1. Implementieren Sie die Funktion `make-subtree-list`, die `make-subtree` auf alle Elemente einer Liste anwendet. Die Elemente der Liste sind entweder `ptree-nodes` oder vom Typ `treenode`, wobei der Inhalt der Wurzel wiederum vom Typ `ptree-node` ist.

2. Schreiben Sie die Funktion `sort-ptree-list`, die eine Liste von Wahrscheinlichkeitsbäumen, jeweils repräsentiert durch ihren Wurzelknoten, nach der Wahrscheinlichkeit sortiert. Orientieren Sie sich an der Lösung der Präsenzübun: Statt Studenten nach der Matrikelnummer, sind Wahrscheinlichkeitsbäumen nach der Wahrscheinlichkeit Ihres Wurzelknotens zu sortieren. Sortieren Sie (`make-subtree-list freq`).

3. Schreiben Sie eine Funktion `build-ptree`. Die Funktion erhält eine Liste von noch zu bearbeitenden Wahrscheinlichkeitsbäumen, repräsentiert durch ihre Wurzelknoten, als Parameter und soll folgendes tun:
   1. Suchen Sie aus allen unbearbeiteten Wahrscheinlichkeitsbäumen, die beiden mit der geringsten Wahrscheinlichkeit im Wurzelknoten aus. Die beiden Wurzelknoten nennen wir t_1 und t_2, wobei die Wahrscheinlichkeit von t_1 kleiner gleich der von t_2 ist.
   2. Löschen Sie t_1 und t_2 aus der Liste der unbearbeiteten Wahrscheinlichkeitsbäumen und fügen Sie einen neuen Wurzelknoten t_3 hinzu. Der linke Sohn von t_3 ist t_1, der rechte ist t_2. Die Wahrscheinlichkeit für t_3 ist die Summe der Wahrscheinlichkeiten von t_1 und t_2. Als Symbol setzen Sie bei t_3 `'unused` ein.
   3. Wiederholen Sie diese Schritte, bis Sie nur noch einen Wahrscheinlichkeitsbaum in der Liste der unbearbeiteten Wahrscheinlichkeitsbäumen finden und liefern Sie diesen zurück.

4. Ändern Sie die Funktion `encode` zu `ptree-encode`, `contains?` zu `ptree-contains?` und `encode-list` zu `ptree-encode-list`, so dass diese mit Wahrscheinlichkeitsbäumen als Codierungsbäumen umgehen können.

5. Codieren Sie die Symbolfolge „informatik macht spass“ mit der `ptree-encode-list` Funktion und dem mit `build-ptree` erzeugten Codierungsbaum. Die Funktion `build-ptree-list` muss dazu mit der Liste (`make-subtree-list freq`) als Parameter aufgerufen werden.

Lösungsvorschlag:

```
1  ;;recursive datastructure for trees
2  (define-struct treenode (left content right))
3
4  ;;structure for the content of ptrees
5  (define-struct ptree-node (s p))
6
7  ;;frequency of letters in english texts
8  (define freq (list
9                (make-ptree-node 'a 0.0651738)
10               (make-ptree-node 'b 0.0124248)
11               (make-ptree-node 'c 0.0217339)
12               (make-ptree-node 'd 0.0349835)
13               (make-ptree-node 'e 0.1041442)
14               (make-ptree-node 'f 0.0197881)
```

```
                 (make-ptree-node 'g 0.0158610)
                 (make-ptree-node 'h 0.0492888)
                 (make-ptree-node 'i 0.0558094)
                 (make-ptree-node 'j 0.0009033)
                 (make-ptree-node 'k 0.0050529)
                 (make-ptree-node 'l 0.0331490)
                 (make-ptree-node 'm 0.0202124)
                 (make-ptree-node 'n 0.0564513)
                 (make-ptree-node 'o 0.0596302)
                 (make-ptree-node 'p 0.0137645)
                 (make-ptree-node 'q 0.0008606)
                 (make-ptree-node 'r 0.0497563)
                 (make-ptree-node 's 0.0515760)
                 (make-ptree-node 't 0.0729357)
                 (make-ptree-node 'u 0.0225134)
                 (make-ptree-node 'v 0.0082903)
                 (make-ptree-node 'w 0.0171272)
                 (make-ptree-node 'x 0.0013692)
                 (make-ptree-node 'y 0.0145984)
                 (make-ptree-node 'z 0.0007836)
                 (make-ptree-node 'space 0.1918182)))


;;a very simple codetree for testing purposes
(define simplecode
  (make-treenode
   (make-treenode
    (make-treenode empty 'a empty)
    'unused
    (make-treenode empty 'b empty))
   'unused
   (make-treenode
    (make-treenode empty 'c empty)
    'unused
    (make-treenode empty 'd empty))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;helper function for turning ptree-nodes into one-element trees.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: ptree-node-or-ptree -> ptree
;;purpose: turns ptree-nodes into trees
;;example: (make-subtree  (make-treenode empty (make-ptree-node 'l 0.0331490) empty) ->
;;                  (make-treenode empty (make-ptree-node 'l 0.0331490) empty)
(define (make-subtree ptree-node-or-ptree)
  (if (treenode? ptree-node-or-ptree) ptree-node-or-ptree (make-treenode empty ptree-node-or-ptree empty)))

;;TEST
(check-expect
 (make-subtree (make-ptree-node 'm 0.0202124))
 (make-treenode empty (make-ptree-node 'm 0.0202124) empty))

;;TEST
```

```
(check-expect
 (make-subtree (make-treenode empty (make-ptree-node 'm 0.0202124) empty)
   )
 (make-treenode empty (make-ptree-node 'm 0.0202124) empty))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.1.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: symbol treenode -> boolean
;;purpose: computes whether symbol is contained in the tree
;;example: (contains? 'a empty) -> false
(define (contains? symbol root)
  (and
   ;;empty tree does not contain symbol
   (not (empty? root))
   (or
    ;;symbol is contained, if it is in root and root is a leaf
    (and (symbol=? symbol (treenode-content root)) (empty? (treenode-left
       root)) (empty? (treenode-right root)))
    ;;or it is contained in the left subtree
    (contains? symbol (treenode-left root))
    ;;or it is contained in the right subtree
    (contains? symbol (treenode-right root)))))

;;TEST
(check-expect
 (contains? 'e simplecode)
 false)

;;TEST
(check-expect
 (contains? 'c simplecode)
 true)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.1.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: symbol tree -> list-of-bit
;;purpose: computes the code for symbol using tree as coding tree
;;example: (encode 'a simplecode) -> '(00)
(define (encode symbol root)
  (cond
    ;;found the symbol, done
    [(symbol=? symbol (treenode-content root)) empty]
    ;;symbol contained in left subtree -> go left prepend 0 to code and
      recurse with left subtree)
    [(contains? symbol (treenode-left root)) (cons '0 (encode symbol (
      treenode-left root)))]
    ;;symbol contained in right subtree -> go right prepend 1 to code and
       recurse with right subtree)
    [else (cons '1 (encode symbol (treenode-right root)))]))

```

```
;;TEST
(check-expect
 (encode 'a simplecode)
 '(0 0))

;;TEST
(check-expect
 (encode 'd simplecode)
 '(1 1))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.1.3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: los tree -> list-of-list-of-bit
;;purpose: computes the code for symbols in los using tree as coding tree
;;example: (encode '(a b) simplecode) -> (list '(00) '(01))
(define (encode-list los root)
  (if (empty? los)
      empty
      (cons (encode (first los) root) (encode-list (rest los) root))))

;;TEST
(check-expect
 (encode-list '(a d) simplecode)
 (list '(0 0) '(1 1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;helper function get-p:
;;contract: ptree -> num
;;purpose: gets the probability from a node of a probability tree
;;example: (get-p (make-treenode empty (make-ptree-node 'a 0.0651738)
  empty) -> 0.0651738
(define (get-p n)
  (ptree-node-p (treenode-content n)))

;;TEST
(check-expect
 (get-p (make-treenode empty (make-ptree-node 'a 123) empty))
 123)

;;some ptrees for testing purposes
(define ptree1 (make-treenode empty (make-ptree-node 'a 0.15) empty))
(define ptree2 (make-treenode empty (make-ptree-node 'b 0.2) empty))
(define ptree3 (make-treenode empty (make-ptree-node 'c 0.3) empty))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.2.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: list-of-ptree-node-or-ptrees
```

```
;;purpose: applies make-subtree to all elements of a list
;;example: (make-subtree-list (list ptree1) -> (list ptree1)
(define (make-subtree-list L)
  (if (empty? L)
      empty
      (cons (make-subtree (first L)) (make-subtree-list (rest L)))))

;;TEST
(check-expect
 (make-subtree-list (list ptree1 (make-ptree-node 'b 0.2)))
 (list ptree1 ptree2))


(check-expect
 (make-subtree-list empty)
 empty)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.2.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: list-of-ptrees -> list-of-ptrees
;;purpose: sorts a list of ptrees by the probability of their root node
;;example:     0.8     0.7              0.7     0.8
;;            /   \   /   \      ->    /   \   /   \
;; copied from exercise 4 (sorting students), adapted the comparison function
(define (sort-ptree-list L)
  (local
    ;;contract: treenode list -> treenode
    ;;purpose: inserts all elements of list into the sorted binary tree root and returns the new sorted binary tree
    ((define (tree-insert-list root L)
       (local
         ;;contract: ptree-node ptree-node -> boolean
         ;;purpose: returns whether the first ptree node's p is <= than the p or the second node
         ;;ONLY CHANGE THIS FUNCTION FROM EXERCISE 4!!!!
         ((define (comp-ptree r1 r2)
            (<= (get-p r1) (get-p r2)))
          ;;contract: treenode num -> treenode
          ;;purpose: inserts content into the sorted binary tree root and returns the new sorted binary tree
          (define (tree-insert root content)
            (cond
              ;; trivial: insert element in the empty tree
              [(empty? root) (make-treenode empty content empty)]
              ;; case1: element must be inserted in left subtree, return new
              ;; tree, where the left son contains the element, content and right son
              ;; remain unchanged
              [(comp-ptree content (treenode-content root))
               (make-treenode
                (tree-insert (treenode-left root) content)
                (treenode-content root)
```

```
                 (treenode-right root))]
               ;; case2: element must be inserted in right subtree, return new
               ;; tree, where the right son contains the element, content and left son
               ;; remain unchanged
               [else (make-treenode
                       (treenode-left root)
                       (treenode-content root)
                       (tree-insert (treenode-right root) content))]))))
          ;;apply tree-insert to all elements of a list
          (if (empty? L)
              root
              (tree-insert-list (tree-insert root (first L)) (rest L)))))
      ;;contract: tree -> list
      ;;purpose:  outputs all elements of the tree in infix order
      ;;example: (flatten-tree empty (tree-insert-list empty '(5 6))) -> '(5 6)
      (define (flatten-tree root)
        (if (empty? root)
            empty
            (append
             (flatten-tree (treenode-left root))
             (list (treenode-content root))
             (flatten-tree (treenode-right root))))))
     ;;insert elements into a binary tree, then output the tree in correct order
     (flatten-tree (tree-insert-list empty L))))


;;TEST
(check-expect
 (sort-ptree-list (list ptree3 ptree2 ptree1))
 (list ptree1 ptree2 ptree3))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.2.3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;contract: list-of-ptrees -> ptree
;;purpose: builds a single ptree from the list of ptrees
;;example: (build-ptree (list ptree1 ptree2)
(define (build-ptree L)
  (if (empty? (rest L))
      ;; only one tree left in the list, so return this tree
      (first L)
      ;; more than one tree left, procede as described
      (local (
               ;;sort the list, so the first two trees in the list have the lowest
               ;;probability
               (define sorted-list (sort-ptree-list L))
               ;; T1 is the tree with the lowest probability
               (define T1 (first sorted-list))
               ;; T2 is the tree with the second lowest probability
               (define T2 (first (rest sorted-list)))
               ;; T3 has T1 and T2 as subtree, content as described
```

```
                (define T3 (make-treenode
                            T1
                            (make-ptree-node
                             'unused
                             (+
                              (get-p T1)
                              (get-p T2)))
                            T2))
                (define list-without-T1-and-T2 (rest (rest sorted-list))))
          ;; recurse with T3 but without T1 and T2
          (build-ptree (cons T3 list-without-T1-and-T2)))))

;; TEST
(check-expect
 (build-ptree (list ptree1 ptree2 ptree3))
 (make-treenode
  ptree3
  (make-ptree-node 'unused 0.65)
  (make-treenode
   ptree1
   (make-ptree-node 'unused 0.35)
   ptree2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TASK 5.2.4
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; contract: symbol ptree -> boolean
;; purpose: returns whether a symbol is contained in a tree
;; example: (ptree-contains? 'a ptreecode) -> true
(define (ptree-contains? symbol root)
  (and
   ;; empty tree does not contain symbol
   (not (empty? root))
   (or
    ;; symbol is contained, if it is in root and root is a leaf (last part is optional)
    (and
     ;; adapted from contains? the symbol must be accessed via ptree-node-s
     (symbol=? symbol (ptree-node-s (treenode-content root)))
     ;; optionally checking if the node is a leaf
     (empty? (treenode-left root))
     (empty? (treenode-right root)))
    ;; or it is contained in the left subtree
    (ptree-contains? symbol (treenode-left root))
    ;; or it is contained in the right subtree
    (ptree-contains? symbol (treenode-right root)))))

;; TEST
(check-expect
 (ptree-contains? 'c ptree3)
 true)

;; TEST
(check-expect
 (ptree-contains? 'c (make-treenode empty (make-ptree-node 'unused 0.9)
```

```
    ptree3))
 true)

;;contract: ptree-encode: symbol ptree -> list-of-bit
;;purpose: computes the code for symbol using ptree as coding tree
;;example: (ptree-encode 'a (make-treenode ptree1 'unused ptree3)) ->
   '(0)
;;adapted from encode the symbol must be accessed via ptree-node-s,
;;use ptree-contains? instead of contains?
(define (ptree-encode symbol root)
  (cond
    ;;found the symbol, done
    [(symbol=? symbol (ptree-node-s (treenode-content root))) empty]
    ;;symbol contained in left subtree -> go left prepend 0 to code and
      recurse)
    [(ptree-contains? symbol (treenode-left root)) (cons '0 (ptree-encode
       symbol (treenode-left root)))]
    ;;symbol contained in right subtree -> go right prepend 1 to code and
       recurse)
    [else (cons '1 (ptree-encode symbol (treenode-right root)))]))

;;TEST
(check-expect
 (ptree-encode 'a (make-treenode ptree1 (make-ptree-node 'unused 0.9)
    ptree3))
 '(0))


;;contract: ptree-encode-list: symbol ptree -> list-of-bit
;;purpose: computes the code for symbols in los using ptree as coding
   tree
;;example: (ptree-encode-list '(a c a) (make-treenode ptree1 'unused
   ptree3)) ->
;;     (list '(0) '(1) '(0))
;;adapted from encode-list, use ptree-encode instead of encode
(define (ptree-encode-list los root)
  (if (empty? los)
      empty
      (cons (ptree-encode (first los) root) (ptree-encode-list (rest los)
         root))))

;;TEST
(check-expect
 (ptree-encode-list '(g d i space i s t space t o l l) (build-ptree (make
   -subtree-list freq)))

 (list
   '(1 0 0 0 1 1)
   '(1 1 0 1 0)
   '(0 1 1 0)
   '(1 1 1)
   '(0 1 1 0)
   '(0 0 1 1)
   '(1 1 0 0)
   '(1 1 1)
   '(1 1 0 0)
   '(1 0 0 1)
   '(1 0 1 1 0)
```

```
    '(1 0 1 1 0)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;TASK 5.2.5
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(ptree-encode-list '(i n f o r m a t i k space m a c h t space s p a s s)
    (build-ptree (make-subtree-list freq)))
```