Grundlagen der Informatik 1

Wintersemester 2009/2010

Prof. Dr. Max Mühlhäuser, Dr. Rößling
http://proffs.tk.informatik.tu-darmstadt.de/teaching

Übung 3 Version: 1.0 02. 11. 2009

---

1 Mini Quiz

Kreuzen Sie die wahren Aussagen an!

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

☐ Eine Baumstruktur kann in Scheme 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. Erläutern Sie Probleme, die entstehen können, wenn Daten redundant abgelegt werden.

3 Nutzung von local

Der RGB-Code eines Farbtons ist ein Tripel (R, G, B) mit $R, G, B \in [0; 255]$. Die Werte R, G und B sind dabei die Anteile der Grundfarben Rot, Grün und Blau am Farbton. Sie reichen von 0 (nicht im Farbton vorhanden) bis 255 (in voller Intensität im Farbton vorhanden). So steht (255 0 0) für reines Rot, (114 247 160) für 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 ist z.B. (204, 102, 153)?
Um dieses Problem zu lösen, kann man den Winkel H (H für hue, Farbton) zu einem RGB-Tripel berechnen. Ein H-Winkel in der Nähe von 0° bedeutet z.B. rötlich, in der Nähe von 240° bedeutet bläulich. Aus Wikipedia stammen folgende Formeln zur Errechnung des Winkels H zu einem RGB Tripel (R, G, B):

$$r = R/255, g = G/255, b = B/255$$
$$M = \max(r, g, b), m = \min(r, g, b)$$
$$H = \begin{cases} 0^\circ & \text{if } M = m, \\ 60^\circ * \frac{g-b}{M-m} & \text{if } r = M, \\ 120^\circ + 60^\circ \times \frac{b-r}{M-m} & \text{if } g = M, \\ 240^\circ + 60^\circ \times \frac{r-g}{(M-m} & \text{if } b = M \end{cases}$$

Falls der nach der Formel berechnete H-Wert negativ ist, ist 360 zu ihm zu addieren.
Scheme nutzt eine Struktur `color` mit den Feldern *red, blue* und *green* zum Speichern von RGB Tripeln.
Anbei finden Sie die Lösung einer Implementierung der *hue*-Funktion, die aus einer als *color*-Struktur übergebenen RBG-Farbe den H-Wert berechnet.

```
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)]
11
12        [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)]))
```

Ihre Aufgabe ist es nun, aus dieser Lösung eine *sinnvolle* Lösung zu machen, indem Sie über **local** gleichlautende Codeteile - Prozeduren oder einmalig berechenbare Werte - *sinnvoll* in *local*-Blöcken berechnen.
Hinweis: Nutzen Sie einen Bleistift und ein Blatt Papier, um zunächst eine Liste der "mehrfach berechneten aber identischen Werte" zu bestimmen. Anhand dieser Liste können Sie dann einfach bestimmen, welche Elemente in einem (oder in mehreren geschachtelten) *local*-Blöcken definiert werden sollten.

4 Tree Sort

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

Hintergrund: Die Datenstruktur sortierter Binärbaum

Ein Binärbaum 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 Baumknoten. Der Inhalt des linken Sohns von n ist entweder ein leerer Baum (empty) oder ein sortierter Binärbaum, dessen größtes Element stets kleiner oder gleich dem Inhalt von n ist. Der Inhalt des rechten Sohns ist entweder ein leerer Baum (empty) oder ein sortierter Binärbaum, dessen kleinstes Element stets größer als der Inhalt von n ist. Vergleichen Sie dazu folgende Scheme-Definition und die Beispiele:

```
1  ;;example binary tree with three nodes
2  ;;     2
3  ;;    / \
4  ;;   1   3
5  ;; / \ / \
6  ;;
7  (make-treenode (make-treenode empty 1 empty) 2
8                  (make-treenode empty 3 empty))
9
10 ;;not a sorted binary tree! Left son is larger than the node content!
11 ;;     2
12 ;;    / \
```

```
;;   4    3
;; /  \  /  \
;;
(make-treenode (make-treenode empty 4 empty) 2
               (make-treenode empty 3 empty))
```

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, siehe T3.30—und machen Sie sich eine Wunschliste von Funktionen. Verwenden Sie `local` für Funktionen, die nicht nach außen sichtbar sein sollen.

1. (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 Binärbaums T' liefert, der alle Elemente von T sowie n enthält. Beachten Sie dabei die Regeln für sortierte binäre Bäume: der linke Sohn muss immer kleiner und der rechte Sohn immer größer als der Inhalt des Vaters sein!

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

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

4. Stellen Sie Ihren Code nun so um, dass lediglich die Prozedur *sort-list* von Außen sichtbar ist, alle Hilfsfunktionen hingegen lokal sind.

5. Angenommen, wir haben folgende Strukturdefinition:

   (**define**−struct student (name matrikel birth−year birth−month birthday))

   Was muss im Code geändert werden, um Studierende nach der *Matrikelnummer* bzw.—in einer separaten Funktion—nach dem *Geburtsjahr* zu sortieren?

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. Falls Sie die Hausübung in einer Lerngruppe bearbeitet haben, geben Sie dies bitte deutlich bei der Abgabe an. Alle anderen Mitglieder der Lerngruppe müssen als Abgabe einen Verweis auf die gemeinsame Bearbeitung einreichen, damit die Abgabe im Portal auch für sie bewertet werden kann.

Abgabedatum: Freitag, 13. 11. 2009, 16:00 Uhr

Denken Sie bitte daran, Ihren Code hinreichend gemäß den Vorgaben zu kommentieren (Scheme: Vertrag, Beschreibung und Beispiel sowie zwei Testfälle pro Funktion; Java: JavaDoc). Zerlegen Sie Ihren Code sinnvoll und versuchen Sie, wo es möglich ist, bestehende Funktionen wiederzuverwenden. Wählen Sie sinnvolle Namen für Hilfsfunktionen und Parameter.
Verwenden Sie als Sprachlevel für die gesamte Hausübung „Zwischenstufe“.

5 Boolesche Bäume (7 P.)

Anstelle der Strukturen *add* und *mul* (→ T3.74ff) betrachten wir hier eine andere Modellierung für boolesche Ausdrücke, die auf *booleschen Bäumen* basiert. Dabei handelt es sich um eine Variante der *Binärbaume* aus Aufgabe 4, bei denen in der Wurzel immer ein *Operator* steht. Wir betrachten für diese Aufgabe zunächst nur die Operatoren *'not* als *unären* sowie *'and, 'or* als *binäre Operatoren*. Gegeben seien folgende Strukturen, die auch in den Vorlagen zu dieser Übung im Portal bereitstehen:

```
;; a bool-input is either
;; 1. a boolean value (true,false,an expression evaluated to true/false)
;; 2. a symbol 'A...'F for a boolean variable
;; 3. (list 'not b), where b is a bool-input
;; 4. (list 'and b1 b2), where b1 and b2 have type bool-input
;; 5. (list 'or b1 b2), where b1 and b2 have type bool-input

;; a bool-tree is either
;; 1. a bool-direct (boolean value or boolean variable)
;; 2. a bool-unary (unary operator, i.e., 'not)
;; 3. a bool-binary (binary operator, e.g., 'and, 'or)

;; a bool-direct represents direct boolean values
;; value: direct-boolean - a boolean value that can be either
;; 1. a boolean value, i.e., something that evaluates to true or false,
;; 2. or a symbol that represents one of the variables 'A...'F
(define-struct bool-direct (value))

;; bool-unary represents unary boolean operators
;; op: symbol - a legal unary operator (e.g., 'not)
;; param: bool-tree - a boolean expression
(define-struct bool-unary (op param))

;; bool-binary represents binary boolean operators
;; op: symbol - a legal binary operator (e.g., 'and, 'or)
;; left: bool-tree - the left (2.) part of the binary boolean expression
;; right: bool-tree - the right (3.) part of the binary boolean expr.
(define-struct bool-binary (op left right))

;; lookup-variable: symbol -> boolean
;; looks up the value of the symbol passed in
;; if undefined, returns an error
;; example: (lookup-variable 'A) is true
(define (lookup-variable variable)
  (cond
    [(symbol=? variable 'A) true]
    [(symbol=? variable 'B) false]
    [(symbol=? variable 'C) true]
    [(symbol=? variable 'D) false]
    [(symbol=? variable 'E) false]
    [(symbol=? variable 'F) true]
    [else (error 'lookup-variable
                 (string-append "Variable "
                                (symbol->string variable)
                                " unknown"))]
    ))
```

Ein bool−input kann also direkt ein boolescher Wert oder boolescher Ausdruck sein, eine der sechs Variablen 'A, ..., 'F oder eine Liste.
Ein bool−expr ist eine Instanz einer der folgenden drei Strukturen:

- (**make**−bool−direct value), wobei value ein boolescher Wert oder eine der Variablen 'A-'F ist,
- (**make**−bool−unary op param), wobei op nur das Symbol '**not** sein darf und param wiederum vom Typ bool−expr ist,

- (**make**−bool−binary op left right), wobei op entweder '**and** oder '**or** ist und left und right wiederum vom Typ bool−expr sind.

Zur Arbeitserleichterung stellt das Template bereits eine Prozedur lookup−variable: **symbol** −> boolean bereit. Diese wertet die Variablen 'A, ..., 'F anhand einer festen Belegung mit entweder **true** oder **false** aus und produziert bei allen anderen Parameterwerten eine Fehlermeldung.
Hinweis: Vergessen Sie für nicht, Vertrag, Zweck, Beispiel und Tests zu jeder global sichtbaren Prozedur zu schreiben! Sie können Hilfsprozeduren als *lokale* Prozeduren deklarieren. Für *lokale* Prozeduren brauchen Sie nur Vertrag, Zweck und Beispiel angeben, aber *keine Tests*. *Um Fehler besser finden zu können, sollten Sie Ihre Prozeduren zunächst nicht lokal definieren und ausführlich testen, bevor Sie sie als* local *redeklarieren!*
Hinweis: Zur Vereinfachung können Sie die Prozeduren **second** und **third** nutzen, die analog zu **first** funktionieren.

1. Schreiben Sie eine Scheme-Prozedur input−>tree: bool−input −> bool−tree. Diese Prozedur konsumiert eine Eingabe vom Typ bool−input wie oben beschrieben und produziert eine (eventuell geschachtelte) Baumstruktur aus bool−tree-Strukturen (2 Punkte).

   Hinweis: Verwenden Sie als Operatoren der booleschen Ausdrücke nur '**not**, '**and** und '**or**.

   Hinweis: Beachten Sie, dass der übergebene *bool-input* fünf verschiedene Formen haben kann und folgen Sie dem Design-Rezept für heterogene Daten!

   Zur besseren Erläuterung der Aufgabe müssen die beiden folgenden Tests bei Ihrem Code erfolgreich sein:

```
1  ;; Tests
2  (check-expect (input->tree (list 'and 'A true))
3                (make-bool-binary 'and
4                                  (make-bool-direct 'A)
5                                  (make-bool-direct true)))
6  (check-expect (input->tree (list 'or (list 'not 'A) 'B))
7                (make-bool-binary 'or
8                                  (make-bool-unary 'not
9                                                   (make-bool-direct 'A))
10                                 (make-bool-direct 'B)))
```

2. Implementieren Sie eine Prozedur eval−unary: **symbol** boolean −> boolean. Diese konsumiert ein Symbol, das für einen unären booleschen Operator steht, sowie einen Wahrheitswert. Als Ergebnis wird die Anwendung des entsprechenden Operators auf den Wahrheitswert produziert (1 Punkt).

   Hinweis: Wir betrachten hier als einzigen unären Operator nur '**not**; bei allen anderen übergebenen Symbolen soll analog zum Vorgehen in lookup−variable ein Fehler produziert werden.

3. Implementieren Sie eine Prozedur eval−binary: **symbol** boolean boolean −> boolean. Diese konsumiert ein Symbol, das für einen binären booleschen Operator steht, sowie zwei Wahrheitswerte. Als Ergebnis wird die Anwendung des entsprechenden Operators auf die Wahrheitswerte produziert (1 Punkt).

   Hinweis: Wir betrachten hier nur die binären Operatoren '**and** und '**or**; alle anderen Werte sollen zu einer Fehlermeldung führen.

4. Implementieren Sie nun eine Prozedur bool−tree−>boolean: bool−tree −> boolean, die einen über input−>tree erzeugten booleschen Baum vollständig auswertet. Nutzen Sie dazu die bereits vorher definierten Hilfsprozeduren (2 Punkte).

5. Beschreiben Sie kurz, wie Sie die Prozeduren schachteln würden, um nur eine nach Außen sichtbare Prozedur eval−input: bool−input −> boolean zu erhalten (1 Punkt).