Informatik I

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ws1617-info1
Abgabestatus/Feedback: https://handin-db.informatik.uni-tuebingen.de

Übungsblatt 4 (11.11.2016)

Abgabe: Freitag 18.11.2016, 14:00 Uhr

Sprachebene „Die Macht der Abstraktion — Anfänger"

1. [2 Punkte] (*Abgabe:* `Blatt04-A1-scopes`)

   Betrachtet das folgende Scheme-Programm.

```
(define x 1)
(define y 5)

((lambda (x y)
    (+ (* 2 x) y))
  y x)

((lambda (a b)
    (+ (* 2 x) y))
  y x)
```

   Gebt die Reduktionsschritte für die beiden Funktionsapplikationen an und erläutert den Begriff der *lexikalischen Bindung* an diesem Beispielprogramm.

2. [4 Punkte] (*Abgabe:* `Blatt04-A2-compact`)

   Kürzt das folgende Programm, in dem ihr überflüssigen Code entfernt oder logische Ausdrücke bzw. nicht notwendige Fallunterscheidungen zusammenfasst. Beachtet, dass die Signaturen der Funktionen nicht verändert werden dürfen. Schreibt eure gekürzten Programme in ein Racket-File und gebt dieses über den Handin-Server ab.

```
(: less-zero? (number -> boolean))
(define less-zero?
  (lambda (x)
    (if (not (< x 0))
        #f
        #t)))

(: f (number -> boolean))
(define f
  ((lambda (x) x)
   (lambda (y)
     (cond
       ((> y 11) #t)
       ((< y 11) #f)
       ((= y 11) #t)))))

(: g (boolean boolean -> boolean))
(define g
  (lambda (a b)
    (or (not b)
        (and a (not a)))))

(: greater-equal-zero? (number -> boolean))
(define greater-equal-zero?
  (lambda (x)
    (cond
      ((>= x 0) #t)
      (else #f))))
```

Hinweis: Die Funktion `(: not (boolean -> boolean))` negiert einen übergebenen Booleschen Wert. Beispiel: `(not #f)` $\rightsquigarrow$ `#t`

3. [4 Punkte] (*Abgabe:* `Blatt04-A3-heiner-or`)

   Lest euch folgende Reduktionsregeln für die Spezialform `or` ($\text{eval}_{\texttt{or}}$) genau durch:

   - (`or` $\langle e_1 \rangle$ $\langle e_2 \rangle \ldots \langle e_n \rangle$):

     (1) Reduziere $\langle e_1 \rangle$, erhalte $\langle e_1' \rangle$

     (2) $\begin{cases} \texttt{\#t} & \text{falls } \langle e_1' \rangle = \texttt{\#t} \ (\text{⚠}\langle e_2 \rangle \ldots \langle e_n \rangle \text{ nicht reduziert}) \\ (\texttt{or } \langle e_2 \rangle \ldots \langle e_n \rangle) & \text{sonst} \end{cases}$

   - (`or`) $\rightsquigarrow$ `#f`

   Löst nun vor dem Hintergrund obiger Reduktionsregeln folgende Aufgabe:

   Heiner Hacker sieht partout nicht ein, warum `or` eine Spezialform sein sollte (abgesehen von der Tatsache, dass `or` eine unbestimmte Anzahl von Argumenten nimmt). Er fragt sich, warum man `or` nicht als eine normale Prozedur definieren kann. Um das herauszufinden, baut Heiner ein `or` mit zwei Argumenten in eine selbstgeschriebene Prozedur nach. Das Ergebnis sieht so aus:

   ```
   (define heiner-or
      (lambda (test-1 test-2)
         (if test-1
             #t
             test-2)))
   ```

   Zu Heiners großer Begeisterung funktioniert `heiner-or` anscheinend genauso wie das eingebaute `or` für zwei Argumente:

   ```
   > (heiner-or (= 10 10) (> 2 5))
   #t
   > (heiner-or (> 23 42) (< 5 2))
   #f
   ```

   Heiners Freundin Eva-Lu Ator ist skeptisch und meint, `heiner-or` verhielte sich anders als das eingebaute `or` für zwei Argumente. Wer von beiden hat Recht? Begründet eure Entscheidung mit dem Substitutionsmodell und der Umformungsregel für `or`.

   Falls Eva-Lu Recht hat: Findet ein Programm, an dem sich der Unterschied beobachten lässt.

4. [6 Punkte] (*Abgabe:* `Blatt04-A4-calendar`)

   Überprüft Kalenderdaten auf ihre Gültigkeit:

   (a) Schreibt eine Daten- und eine Record-Definition für *Kalenderdaten*, welche sich aus der rein numerischen Repräsentation eines Datums (Tag, Monat, Jahr) zusammensetzt.

   (b) Schreibt eine Prozedur `calendar-date-ok?`, die feststellt ob ein Kalenderdatum-Record einem tatsächlichen Kalenderdatum entspricht, also korrekte Daten wie 1.1.1970 von unsinnigen wie 34.17.2016 unterscheidet. Schaltjahre sollen dabei ignoriert werden.

   (c) Schreibt eine Prozedur `calendar-date-ok/leap-year?` die sich wie `calendar-date-ok?` verhält und zusätzlich Schaltjahre berücksichtigt!

   Hinweis: Zur Lösung der Aufgabe kann die eingebaute Prozedur

   ```
   (: modulo (integer integer -> integer))
   ```

   hilfreich sein. Sie bestimmt den Rest einer ganzzahligen Division.

5. [4 Punkte] (*Abgabe:* `Blatt04-A5-soccer`)

   Beim Fußball lässt die Rückennummer eines Spielers häufig Rückschlüsse auf seine Position zu. Wir machen dabei folgende Annahmen:

   - Ein *Torwart* hat die Rückennummer 1.
   - Ein *Abwehrspieler* hat die Rückennummer 2, 3, 4 oder 5.
   - Ein *Mittelfeldspieler* hat die Rückennummer 6, 7, 8 oder 10.
   - Ein *Stürmer* hat die Rückennummer 9 oder 11.
   - Ein *Ersatzspieler* hat eine Rückennummer zwischen 13 und 99.
   - Alle anderen Rückennummern sind ungültig.

   Schreibt nun eine Prozedur mit folgender Signatur:

   ```
   (: player-position (natural -> (one-of "Torwart" "Abwehr" "Mittelfeld" "Sturm" "Ersatz" "Ungültig"))
   ```

   Die Prozedur soll dabei zu einer gegebenen Rückennummer die zugehörige Position berechnen.

   Verwendet beim Schreiben der Prozedur die Konstruktionsanleitungen für Prozeduren und für Fallunterscheidungen. Testet die Prozedur `player-position`, mit mindestens sechs Testfällen, so dass für jede Position (auch ungültige Positionen) ein Testfall existiert und damit alle Fälle abgedeckt sind.