Typische Klausuraufgaben Info 1

[Quelle: Prof. Torsten Grust https://forum-db.informatik.uni-tuebingen.de]
Forenbeiträge von Prof. Grust WS18/19
Zusammengestellt von Jules Kreuer

Klausuraufgabe #1

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-1-medtech/6436]

Ihr kennt `foldr`, die *higher-order function*, die eine gegebene Liste `xs` zu einem Wert `x` "zusammenfalten" kann (und so die Länge, das Maximum/Minimum, die Summe der Elemente in `xs`, … berechnen kann).

Implementiert nun die duale Funktion `unfoldr`, die aus einem einzelnen gegebenen Startwert (*seed value*) `x` eine ganze Liste `xs`erzeugen kann. `unfoldr` besitzt die folgende Signatur:

```
(: unfoldr ((%b -> boolean) (%b -> %a) (%b -> %b) %b -> (list-of %a)))
```

`(unfoldr p f g x)` verhält sich wie folgt:

- Sollte `(p x)` gelten, wird die leere Liste erzeugt. `p` ist also eine Art "Stoppkriterium".
- Ansonsten ist `(f x)` das erste Listenelement. Die Restliste wird aus dem *seed value* `(g x)` erzeugt.

Damit gilt dann z.B.

```
> (unfoldr zero? id (lambda (x) (- x 1)) 10)
#<list 10 9 8 7 6 5 4 3 2 1>
> (unfoldr zero? (lambda (x) (modulo x 10)) (lambda (x) (quotient x 10)) 2019)
#<list 9 1 0 2>
```

Mögliche Lösung (nach Prof. Grust)

```
(: unfoldr ((%b -> boolean) (%b -> %a) (%b -> %b) %b -> (list-of %a)))
```

```
(define unfoldr
   (lambda (p f g x)
      (if (p x)
         empty
         (make-pair (f x) (unfoldr p f g (g x))))))
```

Klausuraufgabe #2

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-2-medtech/6524]

Gegeben sei die folgende rekursive Signatur `(expr-of t)` mittels der **arithmetische Ausdrücke** über Konstanten der Signatur `t`(typischerweise `natural`, `integer` oder `number`) dargestellt werden können:

```
(define expr-of
(lambda (t)
    (signature (mixed
                t                                    ; Konstante
                (add-of (expr-of t) (expr-of t))))))) ; e₁ + e₂
```

```
(: make-add ((expr-of %a) (expr-of %a) -> add))
(define-record-procedures-parametric add add-of
    make-add
    add?
    (add-left add-right))
```

Derzeit kann `expr-of` nur Konstanten und Addition (Ausdrücke der Form `e₁ + e₂`) darstellen.

1. Erweitere `expr-of` so, dass auch Multiplikation (`e₁ * e₂`) dargestellt werden kann.
2. Mittels des so erweiterten `expr-of`, definiere einen Scheme-Ausdruck `e1`, der den arithmetischen Ausdruck `2 + 4 * 10`repräsentiert.
3. Definiere eine Funktion `(: eval ((expr-of %a) -> %a))` die einen gegebenen Ausdruck auswertet und seinen (numerischen) Wert berechnet. Unter anderem soll also `(eval e1) ~~> 42` gelten.

Mögliche Lösung (nach Jules Kreuer)

```
(define expr-of
    (lambda (t)
        (signature (mixed t (add-of (expr-of t) (expr-of t))
                            (mult-of (expr-of t) (expr-of t))))))
```

```
(: make-mult ((expr-of %a) (expr-of %a) -> mult))
(define-record-procedures-parametric mult mult-of
    make-mult
    mult?
    (mult-left mult-right))
```

```
(: e1 (add-of natural (mult-of natural natural)))
(define e1 (make-add 2 (make-mult 4 10)))
```

```
(: eval ((expr-of %a) -> %a))
(define eval
    (lambda (e)
        (match e
              ((make-add l r) (+ (eval l) (eval r)))
              ((make-mult l r) (* (eval l) (eval r)))
              (_ e))))
```

Klausuraufgabe #3

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-3/6576]

Mittels `stream-take` lassen sich endliche Präfixe eines Streams in Listen umwandeln. In dieser Aufgabe betrachten wir die "Umkehrfunktion" `list->stream`, die eine gegebene (endliche) Liste in einen (unendlichen) Stream transformiert.

1. Schreibt die Funktion
2. `(: list->stream ((list-of %a) -> (stream-of (maybe-of %a))))`
3. Wenn die Liste `xs` die Elemente $x_1, x_2, \ldots, x_n$ enthält, dann liefert der generierte Stream `(list->stream xs)` die Elemente $x_1, x_2, \ldots, x_n$, `#f`, `#f`, `#f`, … Nach den ersten *n* Elementen wird also nur noch der Wert `#f` produziert.
4. Welcher Code ist unten für _____ einzusetzen, damit die algebraische Eigenschaft gilt?

```
(check-property
    (for-all ((xs (list-of natural)))
      (expect xs (_______________ (list->stream xs)))))
```

Ihr kennt die Definition der Signatur `(maybe-of t)`, aber hier nochmal zur Erinnerung:

```
(define maybe-of
    (lambda (t)
        (signature (mixed t (one-of #f)))))
```

Mögliche Lösung (nach Prof. Grust)

```
; Turn a list xs into a stream whose prefix are the elements of xs. Following
;these, the stream only contains #f.

(: list->stream ((list-of %a) -> (stream-of (maybe-of %a))))
(define list->stream
    (letrec ((falses (make-cons #f (lambda () falses))))
        (lambda (xs)
            (fold falses
                  (lambda (x str) (make-cons x (lambda () str)))
                  xs))))
```

Klausuraufgabe #4

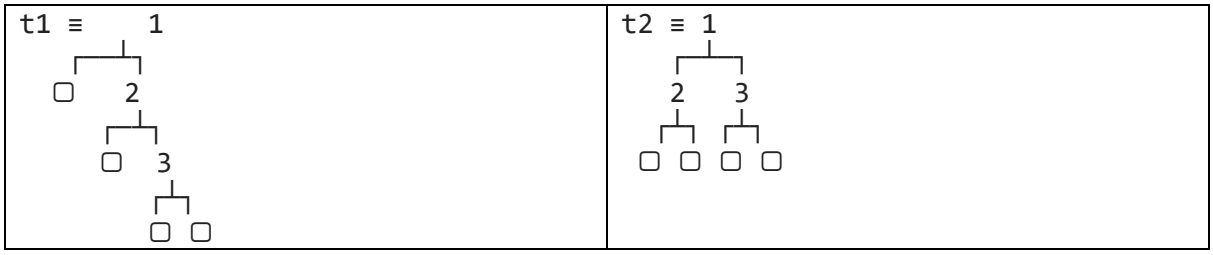
[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-4/6608]

Diese Aufgabe ist von der Art her klausurtypisch, sprengt aber im Zeitaufwand den Rahmen

Definiert ein Prädikat `(: right-deep? ((btree-of %a) -> boolean))`, das für einen Binärbaum `t` feststellt, ob `t` rechts-tief ist. Der leere Baum ist rechts-tief. Es gilt also etwa:

```
(right-deep? t1) ~~> #t
(right-deep? t2) ~~> #f
```

wenn `t1` und `t2` die aus der Vorlesung bekannten Binärbäume bezeichnen:



Wichtig: Implementiert `right-deep?` mittels `btree-fold`!

Der Einsatz eines **trivialen Wrappers** ist erlaubt: Untenstehender Code, in dem `‹trivial›` (offensichtlich eine Funktion der Signatur `(‹sig› -> boolean)`) das eigentliche Ergebnis aus dem Resultat von `right-deep?-worker` extrahiert, geht in Ordnung:

```
(: right-deep? ((btree-of %a) -> boolean))
(define right-deep?
    (lambda (t)
        (‹trivial› (right-deep?-worker t))))

(: right-deep-worker? ((btree-of %a) -> ‹sig›))
(define right-deep?-worker
    (lambda (t)
        (btree-fold ...
                    ...
                    t)))
```

Mögliche Lösung (nach Philipp Hafner)

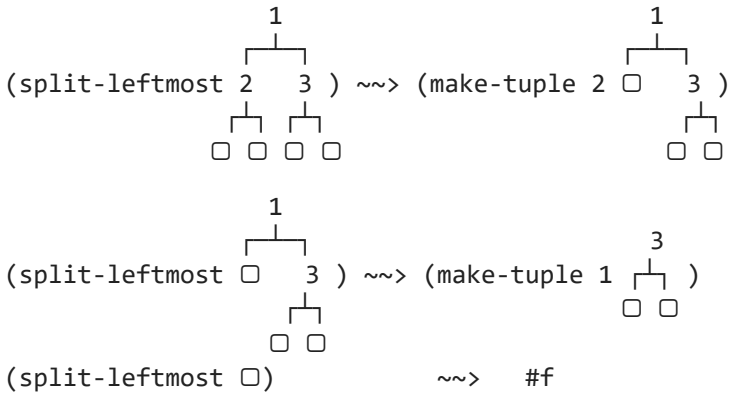
```
(define right-deep?
    (lambda (t)
        (btree-fold (if (empty-tree? t) #t empty-tree)
                    (lambda (left _ right)
                            (and (empty-tree? left)
                                 (or (empty-tree? right) right)))
                     t)))
```

Klausuraufgabe #5

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-5/6637]

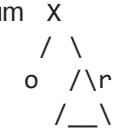
Schreibt die endrekursive Funktion `split-leftmost` mit folgender Signatur (`tuple-of` und `maybe-of` sind so definiert, wie sie euch schon länger bekannt sind):
`(: split-leftmost ((btree-of %a) -> (maybe-of (tuple-of %a (btree-of %a)))))`

`(split-leftmost t)` extrahiert aus Binärbaum `t` den am weitesten links stehenden Knoten `n` und liefert:
1. die Markierung von `n` (der Signatur `%a`) sowie
2. den restlichen Baum (Signatur `(btree-of %a)`), der entsteht, wenn `n` aus `t` entfernt wird.



Mögliche Lösung (nach Prof. Grust)

- Die Lösung sollte endrekursiv sein. Daher nutzt der Worker — unten `split` genannt — einen Akkumulator `k`.
- Für einen Akkumulator ist die Signatur von `k` ungewöhnlich:
  `(:k ((btree-of %a) -> (btree-of %a)))`.
  Der *Akkumulator ist also eine Funktion*.
- Ich nutze diesen Funktion, um einen *Baum mit Anfügeposition* darzustellen, quasi einen Baum mit einem "Loch", in dem ein weiterer Baum platziert werden kann.
  Beispiel: `k ≡ (lambda (o) (make-node o x r))` entspricht dem Baum




```
(: split-leftmost ((btree-of %a) -> (maybe-of (tuple-of %a (btree-of
%a)))))(define split-leftmost
  (letrec ((split
      (lambda (t k)
        (match t
             ((make-empty-tree) #f)
             ((make-node (make-empty-tree) x r) (make-tuple x (k r)))
             ((make-node l x r) (split l (lambda (o) (k (make-node o xr))))))))))
    (lambda (t)
      (split t (lambda (o) o)))))
```

Klausuraufgabe #6

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-6/6681]

Gegeben sei ein Huffman-Tree (wie aus der Vorlesung bekannt, die Definitionen finden sich auch noch einmal unten). Baut die Funktion
`(: huff-encode ((huff-tree-of string) string -> (list-of bit))`
und folgt dabei dem in der Vorlesung erwähnten *Plan B*. Es ergeben sich folgende Teilaufgaben:

1. Schreibt die Funktion `huff-tree-fold`, also eine Fold-Operation für `(huff-tree-of %a)`. Überlegt euch die Signatur für `huff-tree-fold` und gebt auch eine Implementation an.
2. Nutzt `huff-tree-fold` um die Funktion

   `(: huffman-code-table ((huff-tree-of %a)`
   `-> (list-of (tuple-of %a (list-of bit)))))`zu schreiben. `(huffman-code-table ht)` konstruiert eine Lookup-Tabelle, in der alle Zeichen im Huffman-Tree `ht` mit ihrer Bit-Codierung verzeichnet sind (in beliebiger Reihenfolge). Für den Huffman-Tree `code-for-erdbeermarmelade` aus der Vorlesung ergibt sich also beispielsweise:

```
> (huffman-code-table code-for-erdbeermarmelade)
#<list
#<record:tuple "r" #<list 0 0>>
#<record:tuple "m" #<list 0 1 0>>
#<record:tuple "a" #<list 1 1 0>>
#<record:tuple "b" #<list 0 0 0 1>>
#<record:tuple "l" #<list 1 0 0 1>>
#<record:tuple "d" #<list 1 0 1>>
#<record:tuple "e" #<list 1 1>>>
```

3. *Plan B*: Nutzt `huffman-code-table` um eine andere Version von `(huff-encode ht s)` zu bauen, die die Bitfolgen für die Zeichen des String `s` in der Code-Tabelle für den Huffman-Tree `ht` nachschlägt.

Die benötigten Datendefinitionen: [Siehe Anhang 1]

Klausuraufgabe #7

[https://forum-db.informatik.uni-tuebingen.de/t/typische-klausuraufgabe-7-medtech/6737]

Ihr alle kennt (`foldr z c xs`) (bzw. die bereits eingebaute Variante `fold`) auf Listen. Baut die Funktion (`indexed-foldr z c xs`), in der die Funktion `c` nicht nur das aktuell zu verarbeitende Listenelement `x` und das Ergebnis der Faltung auf der Restliste als Parameter bekommt, sondern auch **die Position `i` von `x` in der Liste** (`i` = 0,1,2,…).

Die Signatur lautet:

```
(: indexed-foldr (%b (natural %a %b -> %b) (list-of %a) -> %b))
                          ↑     ↑  ↑
                   Position i, x, Faltung der Restliste
```

Damit ergibt sich dann beispielsweise:

```
(indexed-foldr empty
               (lambda (i x xs) (make-pair (make-tuple i x) xs))
               (list "a" "b" "c"))
~~> #<list #<record:tuple 0 "a"> #<record:tuple 1 "b"> #<record:tuple 2 "c">>
```

und (gewichtete Summe der Elemente der Liste (`list 1 9 0 4`), also 1×1 + 2×9 + 3×0 + 4×4):

```
(indexed-foldr 0 (lambda (i x s) (+ (* (+ i 1) x) s)) (list 1 9 0 4))
~~> 35
```

1. Baut `indexed-foldr` **mittels expliziter Rekursion**. Wenn ihr eine Hilfsfunktion benötigt, nutzt `letrec` um deren Implementation in `indexed-foldr` zu verbergen.

2. Baut `indexed-foldr` **mittels des eingebauten `fold`** ohne Hilfsfunktion.
   (⚠ Kann kniffliger sein.)

Mögliche Lösung (nach Philipp Hafner)

```
; 1) Fold mit Indizes (explizite Rekursion)
(: indexed-foldr (%b (natural %a %b -> %b) (list-of %a) -> %b))
(define indexed-foldr
(lambda (z c xs)
  (letrec [(indexed-foldr-worker
     (lambda (i to-do)
          (match to-do
              [empty z]
              [(make-pair y ys) (c i y (indexed-foldr-worker (+ i 1) ys))])))]
   (indexed-foldr-worker 0 xs))))


; 2) Fold mit Indizes (mittels fold)
(: indexed-foldr2 (%b (natural %a %b -> %b) (list-of %a) -> %b))
(define indexed-foldr2
    (lambda (z c xs)
        ((fold (lambda (_) z)
               (lambda (y ys) (lambda (i) (c i y (ys (+ i 1)))))
                xs) 0)))
```

Anhang

Zu #6

```
; Huffman-Trees
; Ein Blatt eines Huffman-Tree (huff-leaf)
; - trägt eine Markierung (label):
(: make-huff-leaf (%a -> (huff-leaf-of %a)))
(: huff-leaf-label ((huff-leaf-of %a) -> %a))
(define-record-procedures-parametric huff-leaf huff-leaf-of
    make-huff-leaf
    huff-leaf?
    (huff-leaf-label))

; Ein innerer Knoten eines Huffman-Tree (huff-node) besitzt
; - einen linken Teilbaum (left) und
; - einen rechten Teibaum (right):
(: make-huff-node (%a %b -> (huff-node-of %a %b)))
(: huff-node-left ((huff-node-of %a %b) -> %a))
(: huff-node-right ((huff-node-of %a %b) -> %b))
(define-record-procedures-parametric huff-node huff-node-of
    make-huff-node
    huff-node?
    (huff-node-left
    huff-node-right))
; Signatur (huff-tree-of t): Huffman-Tree mit Blättern
; mit Markierungen der Signatur t
(define huff-tree-of
    (lambda (t)
        (signature (mixed (huff-leaf-of t)
                  (huff-node-of (huff-tree-of t) (huff-tree-of t))))))

; Ein Bit eines Zeichencodes
(define bit
(signature (one-of 0 1)))
; ---------------------------------------------------------------------
; Polymorphe Paare
(: make-tuple (%a %b -> (tuple-of %a %b)))
(: tuple? (any -> boolean))
(: tuple-left ((tuple-of %a %b) -> %a))
(: tuple-right ((tuple-of %a %b) -> %b))
(define-record-procedures-parametric tuple tuple-of
    make-tuple
    tuple?
    (tuple-left
    tuple-right))
```