Musterlösung zur 2. Probeklausur – Directors' Cut™
Programmierung 1
Wintersemester 2012/13

Das Team der Tutoren

02. Februar 2013

Dieter Schlau Musterlösung

Name

2442424

Matrikelnummer

Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden.

Prüfen Sie dann zuerst, ob Sie alle **25** Seiten dieser Klausur erhalten haben.

Hilfsmittel sind nicht zugelassen. Am Arbeitsplatz dürfen nur Schreibgeräte, Getränke, Speisen und Ausweise mitgeführt werden. Taschen und Jacken müssen an den Wänden des Klausursaals zurückgelassen werden. Mobiltelefone sind ebenfalls dort ausgeschaltet aufzubewahren.

Das Verlassen des Saals ohne Abgabe des Klausurhefts gilt als Täuschungsversuch.

Wenn Sie während der Bearbeitung zur Toilette müssen, geben Sie bitte Ihr Klausurheft bei der Aufsicht ab. Es kann immer nur eine Person zur Toilette.

Alle Lösungen müssen auf den bedruckten rechten Seiten des Klausurhefts notiert werden. Die leeren linken Seiten dienen als Platz für Skizzen und werden **nicht korrigiert.** Notizpapier ist nicht zugelassen. Sie können mit Bleistift schreiben.

Für die Bearbeitung der Klausur stehen 90 Minuten zur Verfügung. Insgesamt können 90 Punkte erreicht werden. Zum Bestehen der Klausur genügen 45 Punkte.

Bitte legen Sie zur Identifikation Ihren Personalausweis bzw. Reisepass sowie Ihren Studierendenausweis neben sich.

Viel Erfolg!

Bitte lesen
Wir versuchen euch mit dieser Probeklausur eine Vorstellung von Ablauf und Aussehen der Klausur in Programmierung 1 zu geben. Dennoch ist sie, insbesondere was Stoffauswahl und erlaubte Hilfsmittel betrifft, nicht repräsentativ für die Klausur.

Während der Probeklausur dürfen Sie die linken Seiten der Klausur zum Schreiben benutzen.

1	2	3	4	5	6	7	8	9
15	8	14	12	8	15	13	5	10

Summe
$90 + 10$

Note

Hinweise

Folgende Prozeduren sind vordeklariert und dürfen benutzt werden, falls nicht anders angegeben:

- $iter : int \to \alpha \to (\alpha \to \alpha) \to \alpha$
- $first : int \to (int \to bool) \to int$
- $implode : char\ list \to string$
- $map : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$
- $length : \alpha\ list \to int$
- $rev : \alpha\ list \to \alpha\ list$
- $null : \alpha\ list \to bool$
- $hd : \alpha\ list \to \alpha$
- $tl : \alpha\ list \to \alpha\ list$
- $foldl : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$
- $foldr : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$
- $List.concat : \alpha\ list\ list \to \alpha\ list$
- $Char.isAlpha : char \to bool$
- $Vector.sub : \alpha\ vector * int \to \alpha$ $(*Subscript*)$
- $Vector.update : \alpha\ vector * int * \alpha \to \alpha\ vector$

Umgebungen

```
type 'a env = id -> 'a
exception Unbound of id
fun empty x = raise Unbound x
fun update env x a y = if y=x then a else env y

exception Error of string
```

Stapelmaschine

```
type index = int
type noi = int
type noa = int
type sa = int
type ca = int

datatype instruction =
        halt
        | con of int            | new of noa
        | add                   | getH of index
        | sub                   | putH of index
        | mul                   | proc of noa * noi
        | leq                   | arg of index
        | branch of noi         | call of ca
        | cbranch of noi        | return
        | getS of sa            | callR of ca
        | putS of sa
type code = instruction list
```

Sie dürfen Prozeduren aus vorangehenden Aufgaben oder Aufgabenteilen nutzen, selbst wenn Sie diese Aufgabe nicht gelöst haben.

Aufgabe 1: Die Türme von Hanoi

Im Folgenden betrachten wir das Spiel "Die Türme von Hanoi": Das Spiel besteht aus drei Stäben A, B und C, auf die mehrere verschieden groß gelochte Scheiben gelegt werden. Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach B zu versetzen. Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.

Die Lösungsidee lautet wie folgt:

- Wenn nur eine Scheibe auf Turm A liegt:
  Bewege die Scheibe von A nach B.
- Wenn n >1 Scheiben auf Turm A liegen:
  Bewege rekursiv n-1 Scheiben von A nach C.
  Bewege rekursiv eine Scheibe von A nach B.
  Bewege rekursiv n-1 Scheiben von C nach B.

Betrachten Sie nun die entsprechende SML-Prozedur des Algorithmus:

```
fun h (0, (a,b,c)) = []
  | h (1, (a,b,c)) = [(a,b)]
  | h (n, (a,b,c)) = h ((n-1), (a,c,b)) @ h (1, (a,b,c)) @ h ((n-1), (c,b,a))

fun hanoi n = h (n, ("A","B","C"))
```

Diese Prozedur liefert eine Liste mit Tupeln, welche die richtigen Spielzüge darstellen. Beispielsweise bedeutet der Tupel ("A","B"), dass im nächsten Zug die oberste Scheibe von A auf B gelegt wird. Von links nach rechts gelesen liefert diese Liste also eine Anleitung für ein Spiel mit n Scheiben.

In den folgenden Aufgabenteilen geht es nun darum, die Laufzeit und die Korrektheit der Prozedur h, bzw. einer verwandten Prozedur zu analysieren.

Beachten Sie, dass Sie den Algorithmus nicht verstanden haben müssen, um die Aufgabe zu bearbeiten!

Teilaufgabe 1.1 **2 Punkte**
Geben Sie eine Größenfunktion für h an.

$\lambda(n,(a,b,c)) \in \mathbb{N}.n$

Teilaufgabe 1.2 **4 Punkte**

Beschreiben Sie die entsprechende Laufzeitfunktion rekursiv. Vernachlässigen Sie dabei die Nebenkosten von @.

Hinweis: Da zur Veranschlagung der Nebenkosten für @ eine Abschätzung der Größe der Resultatliste von h notwendig wäre, haben wir die Aufgabenstellung vereinfacht und berücksichtigen die von @ versursachten Nebenkosten nicht. Dadurch verändert sich die Laufzeitfunktion, nicht aber die Komplexitätsklasse.

$$
\begin{array}{lcl}
r : \mathbb{N} \to \mathbb{N} & & \\
r\,0 & = & 1 \\
r\,1 & = & 1 \\
r\,n & = & r\,(n-1) + r\,1 + r\,(n-1) + 1 = 2\,r(n-1) + 2
\end{array}
$$

Teilaufgabe 1.3 **3 Punkte**

Bestimmen Sie die Komplexität der Laufzeitfunktion aus Teilaufgabe 1.2 und beweisen Sie Ihre Behauptung mit einem aus der Vorlesung bekannten Rekurrenzsatz.

Es gilt: $g(n) = 2 \in \mathcal{O}(1) = \mathcal{O}(n^0)$

Mit dem exponentiellen Rekurrenzsatz folgt, dass $r \in \mathcal{O}(2^n)$. Konstante Nebenkosten sind in diesem Fall für das asymptotische Laufzeitverhalten irrelevant.

Teilaufgabe 1.4 **5 + 1 Punkte**

Betrachten Sie die Prozedur:

$$
\begin{array}{lcl}
p : \mathbb{N} \to \mathbb{N} & & \\
p\,0 & = & 0 \\
p\,1 & = & 1 \\
p\,n & = & p\,(n-1) + 1 + p\,(n-1) \quad n > 1
\end{array}
$$

Diese berechnet die Anzahl der Spielzüge eines Spiels mit n Scheiben.

Zeigen Sie:

(a) Die Prozedur p berechnet die Funktion $f : \lambda n \in \mathbb{N}.2^n - 1$

Hinweis: Zeigen Sie durch Induktion: Für alle $n \in \mathbb{N} : p\,n = 2^n - 1$

Es gibt mehrere Möglichkeiten, wie man diese Aussage beweisen kann.

- Beweis durch Induktion: Wir unterscheiden drei Fälle:

  Fall 1: $n = 0$

  $p\,0 = 0 = 1 - 1 = 2^0 - 1 = 2^n - 1$

  Fall 2: $n = 1$

  $p\,1 = 1 = 2 - 1 = 2^1 - 1 = 2^n - 1$

  Fall 3: $n > 1$

  Induktionsvoraussetzung: Für alle $m \leq n : p\,m = 2^m - 1$

  Es gilt:

  $$
  \begin{array}{lcll}
  p\,n & = & p\,(n-1) + 1 + p\,(n-1) & \text{Definition } p \\
  & = & 2\,p\,(n-1) + 1 & \text{Arithmetik} \\
  & = & 2 \cdot (2^n - 1 - 1) + 1 & \text{IV mit } m = n - 1 < n \\
  & = & 2^n - 2 + 1 & \text{Arithmetik} \\
  & = & 2^n - 1 & \text{Arithmetik}
  \end{array}
  $$

  ■

- Beweis mit Korrektheitssatz: Wir wollen den Korrektheitssatz anwenden.
  Zu zeigen:
  (i) $Dom\ f = Dom\ p$
  Dazu geben wir die natürliche Terminierungsfunktion von p an: $\lambda n \in \mathbb{N}.n$
  (ii) f erfüllt die definierenden Gleichungen von p
  $f\ 0 = 2^0 - 1 = 1 - 1 = 0$

  $f1 = 2^1 - 1 = 2 - 1 = 1$

$$\begin{aligned} f\ n &= 2^n - 1 \\ &= 2^n - 2 + 1 \\ &= 2 \cdot (2^(n-1) - 1) + 1 \\ &= 2 \cdot f\ (n-1) + 1 \\ &= f\ (n-1) + 1 + f\ (n-1) \end{aligned}$$

(b) $f \in \mathcal{O}(2^n)$

Es gilt: $fn = 2^n - 1 < 2^n = 1 \cdot 2^n \Rightarrow f \in \mathcal{O}(2^n)$

Aufgabe 2: Induktion

Teilaufgabe 2.1 (*...natürlich*) **5 Punkte**
Beweisen Sie die folgende Behauptung durch natürliche Induktion:

$$\forall n \in \mathbb{N} : 7^{2n} - 2^n \text{ ist durch 47 teilbar}$$

Hinweis: Ein Zahl a heißt teilbar durch eine Zahl b, wenn eine Zahl k existiert, sodass: $a = b \cdot k$, $a, b, k \in \mathbb{Z}$.

Wir unterscheiden zwei Fälle:

Fall 1: Sei $n = 0$.

$7^0 - 2^0 = 0 = 47 \cdot 0$

Also ist $7^{2n} - 2^n$ durch 47 teilbar (mit $k = 0$).

Fall 2: Sei $n > 0$.

Induktionsvoraussetzung: $\forall m \in \mathbb{N} : m < n \implies 7^{2m} - 2^m$ ist durch 47 teilbar

$$\begin{aligned} 7^{2n} - 2^n &= 7^{2(n-1)} \cdot 49 - 2^{n-1} \cdot 2 \\ &= 2(7^{2(n-1)} - 2^{n-1}) + 47 \cdot 7^{2(n-1)} \\ &= 2 \cdot 47 \cdot k + 47 \cdot 7^{2(n-1)} && \text{Induktionsvoraussetzung für } n-1 \\ &= 47 \cdot (2 \cdot k \cdot 7^{2(n-1)}) \\ &= 47 \cdot k' \end{aligned}$$

Also ist $7^{2n} - 2^n$ durch 47 teilbar (mit $k = k'$).

Teilaufgabe 2.2 (*...verstärkt*) **3 Punkte**
Betrachten Sie folgende Deklaration einer Prozedur $count : int\ list \to int$, die die Anzahl Einsen in einer Liste xs zählt:

```
fun count xs = foldl (fn (x,a) => if x=1 then a+1 else a) 0 xs
```

Wir wollen nun zeigen, dass *count* korrekt arbeitet. Dazu definieren wir die Funktion $\|xs\|$, die die Anzahl Einsen einer Liste xs darstellt. Damit ergibt sich folgende zu beweisende Aussage:

$$\forall xs \in \mathcal{L}(X) : \|xs\| = foldl(fn(x,a) \Rightarrow if\ x = 1\ then\ a+1\ else\ a)\ 0\ xs$$

Diese Aussage eignet sich nicht für einen Induktionsbeweis, da der Startwert nicht allgemein genug ist.

Verstärken Sie die Korrektheitsaussage zu einer Form, die induktiv beweisbar ist.

Hinweis: Sie müssen den Beweis in dieser Aufgabe nicht führen.

$\forall xs \in \mathcal{L}(X) : \forall k \in \mathbb{N} : \|xs\| + k = foldl(fn(x,a) \Rightarrow if\ x = 1\ then\ a+1\ else\ a)\ k\ xs$

Teilaufgabe 2.3 (*...strukturell*) **0 Punkte**
Beweisen Sie die Aussage aus Aufgabenteil 2 nun durch strukturelle Induktion über $xs \in \mathcal{L}(X)$.

Wir unterscheiden 2 Fälle:

- Fall 1: Sei $xs = nil$.

  Dann gilt: $\|xs\| + k = \|nil\| + k = k$

  $$\begin{array}{lll} & foldl(fn(x,a) \Rightarrow\ if\ x = 1\ then\ a+1\ else\ a)\ k\ xs & \\ = & foldl(fn(x,a) \Rightarrow\ if x = 1\ then\ a+1\ else\ a)\ k\ nil & xs = nil \\ = & k & \text{Definition } foldl \end{array}$$

- Fall 2: Sei $xs = x :: xr$.

  Wir unterscheiden erneut über zwei Fälle:

  - Für $x = 1$ gilt:

    $$\begin{array}{lll} & foldl(fn(x,a) \Rightarrow\ if x = 1\ then\ a+1\ else\ a)\ k\ xs & \\ = & foldl(fn(x,a) \Rightarrow\ if\ x = 1\ then\ a+1\ else\ a)\ k\ (x :: xr) & \\ = & foldl(fn(x,a) \Rightarrow\ if\ x = 1\ then\ a+1\ else\ a)\ (k+1)\ xr & \text{Definition } foldl \\ = & \|xr\| + k + 1 & \text{Induktionsvoraussetzung für } xr \\ = & \|x :: xr\| + k & \\ = & \|xs\| + k & \end{array}$$

  - Für $x \neq 1$ gilt:

    $$\begin{array}{lll} & foldl(fn(x,a) \Rightarrow\ if x = 1\ then\ a+1\ else\ a)\ k\ xs & \\ = & foldl(fn(x,a) \Rightarrow\ if\ x = 1\ then\ a+1\ else\ a)\ k\ (x :: xr) & \\ = & foldl(fn(x,a) \Rightarrow\ if\ x = 1\ then\ a+1\ else\ a)\ k\ xr & \text{Definition } foldl \\ = & \|xr\| + k & \text{Induktionsvoraussetzung für } xr \\ = & \|x :: xr\| + k & \\ = & \|xs\| + k & \end{array}$$

Aufgabe 3: Laufzeit

Teilaufgabe 3.1 (*Wer lesen kann, ist klar im Vorteil!*) **2 + 3 + 0 + 0 + 3 Punkte**
Geben Sie die Komplexitäten der folgenden *mathematischen* Objekte an! Geben Sie, sofern notwendig, auch eine geeignete Größen- und eine dazu passende Laufzeitfunktion an!

Tipp: $x \in \mathbb{N} \to \mathbb{N}$ stellt eine *Funktion* dar, $x : \mathbb{N} \to \mathbb{N}$ stellt eine mathematische *Prozedur* dar.

(a) $f \in \mathbb{N} \to \mathbb{N}$
$f\,n = if\ n = 0\ then\ 1\ else\ 2 \cdot f(n-1)$

Wie man leicht einsehen, oder mit dem exponentiellen Rekurrenzsatz folgern kann, gilt $f \in \mathcal{O}(2^n)$. Da f eine Funktion und keine Prozedur ist, kann man keine Größen- bzw. Laufzeitfunktion aufstellen, denn diese haben wir nur für Prozeduren definiert.

(b) $p : \mathbb{N} \to \mathbb{N}$
$$\begin{array}{lcll} p\,0 & = & 1 & \\ p\,n & = & p\,(n-1) & n > 0 \end{array}$$

- Größenfunktion $\lambda n \in \mathbb{N}.n$
- Laufzeitfunktion:
  $g : \mathbb{N} \to \mathbb{N}_+$
  $$\begin{array}{lcll} g\,0 & = & 1 & \\ g\,n & = & g\,(n-1) + 1 & n > 0 \end{array}$$
- Komplexität (laut polynomiellem R.S.): $\mathcal{O}(n)$

(c) $q : \mathbb{Z} \to \mathbb{N}$
$$\begin{array}{lcll} q\,0 & = & 1 & \\ q\,n & = & q\,(n-1) + p\,n & n > 0 \\ q\,n & = & q\,(n+1) + p(-n) & n < 0 \end{array}$$

- Größenfunktion $\lambda z \in \mathbb{Z}.|z|$
- Laufzeitfunktion:
  $r : \mathbb{N} \to \mathbb{N}_+$
  $$\begin{array}{lcll} r\,0 & = & 1 & \\ r\,n & = & r(n-1) + g(n) + 1 & n > 0 \end{array}$$
- Komplexität (laut polynomiellem R.S.): $\mathcal{O}(n^2)$

(d) $\lambda n \in \mathbb{N}.234$

Komplexität (aus ähnlichen Gründen wie oben): $\mathcal{O}(234) = \mathcal{O}(1)$

(e) $h : \mathbb{N} \to \mathbb{N}$
$$\begin{array}{lcll} h\,0 & = & 9 & \\ h\,1 & = & 90 & \\ h\,2 & = & 42 & \\ h\,3 & = & 0 & \\ h\,4 & = & 234624562389897 & \\ h\,n & = & h\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + 789 & n \geq 5 \end{array}$$

- Größenfunktion $\lambda n \in \mathbb{N}.n$

- Laufzeitfunktion:
  $s : \mathbb{N} \to \mathbb{N}_+$
  $s\, n \quad = \quad 1 \qquad\qquad n < 5$
  $s\, n \quad = \quad s\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + 1 \quad n \geq 5$
- Komplexität (laut logarithmischem R.S.): $\mathcal{O}(log\ n)$

Teilaufgabe 3.2 (*Auf die Größe kommt es an!*) **3 + 3 + 0 Punkte**
Bestimmen Sie die Komplexität der folgenden *Prozeduren*, gemäß der angegebenen Größenfunktionen, indem Sie die Laufzeitfunktionen bestimmen!

(a) $@ : \mathcal{L}(\mathcal{X}) \times \mathcal{L}(\mathcal{X}) \to \mathcal{L}(\mathcal{X})$
$nil@ys \quad = \quad ys$
$(x :: xr)@ys \quad = \quad x :: (xr@ys)$

Größenfunktionen:

(i) $\lambda(xs, ys).3|xs| + 7$

Laufzeitfunktion:
$r : \mathbb{N} \to \mathbb{N}_+$
$r\, n \quad = \quad 1 \qquad\qquad n \leq 7$
$r\, n \quad = \quad r\,(n-3) + 1 \quad n > 7$
Komplexität (laut polynomiellem R.S.): $\mathcal{O}(n)$

(ii) $\lambda(xs, ys).2^{|xs|}$

Laufzeitfunktion:
$r' : \mathbb{N} \to \mathbb{N}_+$
$r'\, n \quad = \quad 1 \qquad\qquad n \leq 1$
$r'\, n \quad = \quad r'\left(\frac{n}{2}\right) + 1 \quad n > 1$
Komplexität (laut logarithmischem R.S.): $\mathcal{O}(log\ n)$
Eine ausführliche Erklärung hierzu findet sich auch unter
`http://www.ps.uni-saarland.de/courses/prog-ws12-forum/viewtopic.php?f=5&t=210`

(b) $sauschwer : \mathbb{N} \to \mathbb{N}$
$sauschwer\ 0 \quad = \quad 78$
$sauschwer\ n \quad = \quad sauschwer\ (n-1) \quad n > 0$

Größenfunktion: $\lambda n.\lceil log(n+1) \rceil$

Hinweis: Versuchen Sie *nicht*, die Laufzeitfunktion rekursiv anzugeben!

Laufzeitfunktion: $r : \mathbb{N} \to \mathbb{N}_+, r(n) = 2^n$

Es ist trivial, einzusehen, dass die Laufzeit von *sauschwer* linear ist. Als Laufzeitfunktion ergibt sich nämlich $\lambda n.n + 1$.

Die gegebene Größenfunktion soll unser Leben aber interessanter machen! Um sie zu verstehen, schreibe man sich einige Funktionswerte in einer Tabelle auf:

$0 \to 0$
$1 \to 1$
$2 \to 2$
$3 \to 3$
$4 \to 3$
$5 \to 3$
$6 \to 3$
$7 \to 3$
$8 \to 4$
$9 \to 4$
$\vdots$

Um nun für eine Argumentgröße n die zugehörige Laufzeit r(n) zu bestimmen, muss man das Argument mit der Größe n finden, dass die größte Laufzeit hat. Beispiel: Für die Argumentgröße $n = 3$ gibt es nur die Argumente 3, 4, 5, 6 und 7. Da wir uns schon klar gemacht haben, dass die Laufzeit von *sauschwer*, wenn man ehrlich ist, linear ist, muss 7 das Argument mit der größten Laufzeit sein. Allgemein gilt $rn = 2^n - 1 + 1$. Die -1 lässt sich wieder für $n = 3$ einsehen, denn $2^3 - 1 = 7$. Die $+1$ entsteht dadurch, dass die Laufzeit von *sauschwer* für ein Argument x immer $x + 1$ ist.

Komplexität $\mathcal{O}(2^n)$

Hieran wird einmal mehr deutlich, dass man sich für eine Prozedur fast beliebige Laufzeiten zurechtschummeln kann, wenn man nur eine möglichst abartige Größenfunktion wählt!

Teilaufgabe 3.3 (*Ein alter Bekannter*) **0 Punkte**

Untenstehend finden Sie zu Ihrer großen Begeisterung noch einmal den Code für das Sortieren durch Mischen. Zeigen Sie, dass Sortieren durch Mischen bezüglich der Größe der zu sortierenden Liste die Komplexität $\mathcal{O}(n \cdot log\ n)$ hat, indem Sie die Teilprozeduren analysieren!

Achten Sie auf korrekte und sinnvolle Größenfunktionen!

Sie dürfen als gegeben annehmen, dass $foldl$ die Komplexität $\mathcal{O}(n)$ bezüglich der Länge der zu faltenden Liste hat!

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys)) (nil, nil) xs
```

Größenfunktion: $\lambda xs.|xs|$. Da $foldl$ lineare Laufzeit hat und der Operator :: keine Nennenswerten Nebenkosten verursacht, ist $\mathcal{O}(n)$ als Komplexität naheliegend.

```
fun merge (nil , ys) = ys
|   merge (xs , nil ) = xs
|   merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                                   else y::merge(x::xr,yr)
```

Größenfunktion: $\lambda(xs, ys).|xs| + |ys|$.

Laufzeitfunktion:

$$r_m : \mathbb{N} \to \mathbb{N}_+$$
$$r_m\ 0 = 1$$
$$r_m\ n = r_m(n-1) + 1 \qquad n > 0$$

Bei einem Argument der Größe n kann es sich zwar auch um ein Tupel handeln, dass eine Liste der Länge n und eine leere Liste enthält, für dass sich dann Laufzeit 1 ergäbe. Allerdings liegt der Laufzeitfunktion immer die Worst-Case-Annahme zu Grunde (siehe Abschnitt 11.1, ganz unten).

Komplexität (laut polynomiellem R.S.): $\mathcal{O}(n)$.

```
fun msort [] = []
|   msort [x] = [x]
|   msort xs = let val (ys,zs) = split xs
               in merge(msort ys, msort zs) end
```

Größenfunktion: $\lambda xs.|xs|$.

Laufzeitfunktion (r_s bezeichnet die Laufzeitfunktion von *split*):

$$r : \mathbb{N} \to \mathbb{N}_+$$
$$r\ 0 = 1$$
$$r\ 1 = 1$$
$$r\ n = r_s(n) + r\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + r\left(\left\lceil \frac{n}{2} \right\rceil\right) + r_m(n) + 1 \qquad n > 1$$

Die Rekurrenz wird klar, wenn man sich verdeutlicht, was msort für nicht triviale Argumente tut. Hierbei muss man sich klar machen, dass split zwei Listen (ungefähr) halber Länge zurückliefert. Der Summand

$r_m(n)$ kommt deshalb zustande, weil das Sortieren der Teillisten deren Länge nicht verändert. Man kann die Rekurrenz dieser Funktion auch etwas vereinfachen:

$$r\ n = r\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + r\left(\left\lceil \frac{n}{2} \right\rceil\right) + g(n)$$

wobei $g(n) := r_s(n) + r_m(n) + 1$. Wie sich aus der Analyse von *split* und *merge* ergibt, ist $g(n) \in \mathcal{O}(n)$.

Leider müssen wir an dieser Stelle ein bischen schummeln: Die beiden anderen Summanden der Rekurrenz fassen wir etwas salopp folgendermaßen zusammen:

$$r\ n = 2 \cdot r\left(\frac{n}{2}\right) + g(n)$$

Das ist zwar nur dann vollständig korrekt, wenn die übergebene Liste eine gerade Länge hat, aber der Fehler, den wir dabei für Listen ungerade Länge machen, ist hinreichend gering. Eine solche Schummelei ist zwar nicht schön und wäre in der echten Klausur sehr wahrscheinlich nicht nötig, aber das heißt nicht, dass Aufgaben wie diese dort nicht auftauchen können! Wir wollen euch auf das schlimmste vorbereiten!

Jedenfalls hat die Rekurrenz jetzt genau die richtige Form, um mit dem linear-logarithmischen Rekurrenzsatz auf die Komplexität $\mathcal{O}(n \cdot log\ n)$ schließen zu können.

Aufgabe 4: Statische und dynamische Semantik

Wir möchten die statische und dynamische Semantik von F um Listen erweitern, um so auch schöne Prozeduren wie $foldl$ in F benutzen zu können. Dazu erweitern wir die abstrakte Syntax unserer neuen Sprache F folgendermaßen um Listen und die zugehörigen Operatoren :: und @:

```
datatype con = False | True | IC of int | Nil    (* constants *)
type     id  = string                            (* identifiers *)
datatype opr = Add | Sub | Mul | Leq | Append    (* operators *)
datatype ty  =                                   (* types *)
   ...
  | List  of ty                                  (* list type *)
  | Alpha                                        (* list type for nil *)
datatype exp =                                   (* expressions *)
   ...
  | App  of exp * exp                            (* procedure application *)
  | Cons of exp * exp                            (* list construction *)
```

Hinweis: Wir erlaubten die Konstruktion von Listen nur mit dem ::-Operator. Ausdrücke der Form **[a,b,c]** liefern also schon in der lexikalischen Analyse einen Fehler. Erlaubt sind jedoch Ausdrücke der Form **a::b::c::nil**.

Teilaufgabe 4.1 (*Statische Semantik - Inferenzregeln*) **6 Punkte**
Ergänzen Sie die statische Semantik der Sprache F' um die entsprechenden Regeln für Ausdrücke mit Listen! Gehen Sie davon aus, dass *nil*, `::` und der *append*-Operator analog zu SML getypt werden und vervollständigen Sie unter dieser Voraussetzung die folgenden Inferenzregeln:.

(a) `nil`

$$\textbf{Snil}\ \frac{}{T \vdash nil :}$$

(b) `::` (Der *cons*-Operator)

$$\textbf{Sconsn}\ \frac{T \vdash e_2 : \alpha\, list}{T \vdash e_1 :: e_2 :} \qquad \textbf{Scons}\ \frac{T \vdash e_2 : t\, list}{T \vdash e_1 :: e_2 :}$$

(c) `@` (Der *append*-Operator)

$$\textbf{Soaln}\ \frac{T \vdash e_2 : \alpha\, list}{T \vdash e_1 @ e_2 :} \qquad \textbf{Snoal}\ \frac{T \vdash e_1 : \alpha\, list}{T \vdash e_1 @ e_2 :}$$

$$\textbf{Soal}\ \frac{}{T \vdash e_1 @ e_2 :}$$

(a) `nil`

Die Konstante `nil` repräsentiert die leere Liste und hat standardmäßig den Typ $\forall\alpha.\,\alpha\,list$. Ohne Kontext kann man keine genauere Aussage über α treffen, es ist also (noch) weder *bool* noch *int*.

$$\textbf{Snil}\quad \frac{}{T \vdash nil : \alpha\,list}$$

(b) `::` (Der *cons*-Operator)

Der Operator `::` wird zum Konstruieren einer Liste benutzt und folgt folgendem Typschema:

$$\forall\alpha.\,\alpha * \alpha\,list \to \alpha\,list$$

Bezüglich der statischen Semantik von F' ist hierbei zu beachten, dass ein Ausdruck, der den `::`-Operator enthält niemals den Typ $\alpha\,list$ haben kann, sondern durch den Typ des linken Teilausdrucks eindeutig bestimmt ist. Ferner ist hierbei zu beachten, dass eine Liste stets nur Elemente des **gleichen** Typs beinhalten kann.

$$\textbf{Sconsn}\quad \frac{T \vdash e_1 : t \qquad T \vdash e_2 : \alpha\,list}{T \vdash e1 :: e2 : t\,list} \qquad \textbf{Scons}\quad \frac{T \vdash e_1 : t \qquad T \vdash e_2 : t\,list}{T \vdash e_1 :: e_2 : t\,list}$$

(c) `@` (Der *append*-Operator)

Den Operator `@` nutzen wir zum Konkatenieren von Listen. Er folgt dem Typschema:

$$\forall\alpha.\,\alpha\,list * \alpha\,list \to \alpha\,list$$

$$\textbf{Snoal}\quad \frac{T \vdash e_1 : \alpha\,list \qquad T \vdash e_2 : t\,list}{T \vdash e_1 @ e_2 : t\,list} \qquad \textbf{Soaln}\quad \frac{T \vdash e_1 : t\,list \qquad T \vdash e_2 : \alpha\,list}{T \vdash e_1 @ e_2 : t\,list}$$

$$\textbf{Soal}\quad \frac{T \vdash e_1 : t\,list \qquad T \vdash e_2 : t\,list}{T \vdash e_1 @ e_2 : t\,list}$$

Teilaufgabe 4.2 (*elab*) **0 Punkte**

Vervollständigen Sie folgenden Elaborierer für F'.

```
fun elabCon True   =

  | elabCon False  =

  | elabCon (IC _) =

  | elabCon Nil    =

fun elabOpr Add Int Int =

  | elabOpr Sub Int Int =

  | elabOpr Mul Int Int =

  | elabOpr Leq Int Int =
```

```
  | elabOpr Append (List t) (List t') =



  |



fun elab f (Con c)          = elabCon c

  | elab f (Id x)           = f x

  | elab f (Opr(opr,e1,e2)) = elabOpr opr (elab f e1) (elab f e2)

  | elab f (If(e1,e2,e3))   = (case (elab f e1, elab f e2, elab f e3) of
                                    (Bool, t2, t3) => if t2=t3 then t2
                                                      else raise Error "T If1"
                                  | _ => raise Error "T If2")

  | elab f (Abs(x,t,e))     = Arrow(t, elab (update f x t) e)

  | elab f (App(e1,e2))     = (case elab f e1 of
                                  Arrow(t,t') => if t = elab f e2 then t'
                                                 else raise Error "T App1"
                                | _  => raise Error "T App2")

  | elab f (Cons(e1,e2))    =
```

```
fun elabCon ... = ...
  | elabCon Nil = List Alpha
fun elabOpr ... = ...
  | elabOpr Append (List t) (List t') =  case (t , t') of
                                           (t , Alpha)  => List t
                                         | (Alpha , t') => List t'
                                         | (t , t')     => if t = t' then List t
                                                           else raise Error "T Opr1"
                                         | _            => raise Error "T Opr"
fun elab f ... = ...
  | elab f (Cons(e1,e2)) = case (elab f e1 , elab f e2) of
                             (t , Alpha)   => List t
                           | (t , List t') => if t = t' then List t
                                              else raise Error "T Cons1"
                           | _             => raise Error "T Cons2"
```

Teilaufgabe 4.3 (*Dynamische Semantik - Evaluierer*) **6 Punkte**

Nun widmen wir uns der Evaluierung. Hierbei konstruieren wir Listen, wie aus Kapitel 4.1 im Buch bekannt, mit Hilfe von Tupeln. Die leere Liste wird durch das nullstellige Tupel dargestellt. Beispielsweise entspricht die Liste $[1, 2]$ dem Tupel $(1, (2, ()))$. Dafür erweitern wir den Datentyp *value* folgendermaßen:

```
datatype value =
    IV    of int
  | Proc  of id * exp * value env
  | LV    of value * value          (* list values *)
  | NV                              (* nil value *)
```

Die Liste $[1, 2]$ wird also durch den Term `LV ( IV 1 , LV (IV 2,NV) )` dargestellt.

Die dynamische Semantik von F wird durch folgende Inferenzregeln auf die von F' erweitert:

$$\textbf{Dnil}\ \frac{}{V \vdash nil \rhd ()} \qquad \textbf{Dcons}\ \frac{V \vdash e_1 \rhd v \qquad V \vdash e_2 \rhd v'}{V \vdash e_1 :: e_2 \rhd (v, v')}$$

$$\textbf{Dnoal}\ \frac{V \vdash e_1 \rhd () \qquad V \vdash e_2 \rhd v}{V \vdash e_1 @ e_2 \rhd v} \qquad \textbf{Doal}\ \frac{V \vdash e_1 \rhd (v, v') \qquad V \vdash e_2 \rhd v'' \qquad z = (v, v' @ v'')}{V \vdash e_1 @ e_2 \rhd z}$$

Vervollständigen Sie nun den Evaluierer für F' in SML, indem Sie die rechten Seiten der folgenden Deklaration ergänzen.

```
fun evalCon False  = IV 0
   ...
  | evalCon Nil    =



fun evalOpr Add (IV x1) (IV x2) =
   ...
  | evalOpr Append                  =

  | evalOpr Append                  =
   ...


fun eval f (Con c)          = evalCon c
  | eval f (Opr(opr,e1,e2)) = evalOpr opr (eval f e1) (eval f e2)
   ...
  | eval f (Cons(e1,e2))    =
```

```
fun evalCon ... = ...
  | evalCon Nil = NV
fun evalOpr ... = ...
  | evalOpr Append  NV         ys = ys
```

```
  | evalOpr Append (LV (x,xr)) ys = LV (x, evalOpr Append xr ys)
  | evalOpr  _  _  _              = raise Error "R Opr"
fun eval f ... = ...
  | eval f (Cons(e1,e2))    = LV ( eval f e1 , eval f e2 )
```

Aufgabe 5: Konkrete Syntax

Teilaufgabe 5.1 (*Lexer für boolesche Ausdrücke*) **8 Punkte**
Gegeben sei der folgende Datentyp:

```
datatype token = AND | OR | TRUE | FALSE | ID of string
exception Error of string
```

Boolesche Ausdrücke bestehen aus t, f, $\wedge$, $\vee$ und Bezeichnern. Die Bezeichner sind Zeichenketten, in denen *nur* Groß- und Kleinbuchstaben vorkommen dürfen, jedoch keine Ziffern oder Sonderzeichen. Bezeichner beginnen außerdem nie mit t oder f. Ihre Prozedur soll also beispielsweise aus „five“ zu $[FALSE, ID''ive'']$ lexen.

Die Token **AND** und **OR** stellen wir durch die Zeichen „$\wedge$“ und „$\vee$“ dar.

Bei ungültigen Bezeichnern oder unzulässigen Zeichen soll eine Ausnahme geworfen werden. Die einzelnen Wörter werden nur durch Leerzeichen getrennt, nicht durch Zeilenumbrüche oder ähnliche Sonderzeichen.

Beispiel:

```
lex (explode "t ∧ bla ∨ blub")
[TRUE, AND, ID "bla", OR, ID "blub"]
```

Schreiben Sie eine Prozedur *lex* : *char list* $\rightarrow$ *token list*, die boolesche Ausdrücke in eine Tokenliste umwandelt.

```
fun lex nil = nil
  | lex (#" ":: cr) = lex cr
  | lex (#"∧":: cr) = AND:: lex cr
  | lex (#"∨":: cr) = OR:: lex cr
  | lex (#"t"::cr) = TRUE:: lex cr
  | lex (#"f"::cr) = FALSE:: lex cr
  | lex (c::cr) =  if Char.isAlpha c then lexId [c] cr
                   else raise Error "lex"
and lexId cs csX = if null csX orelse not(Char.isAlpha (hd csX))
                   then ID(implode(rev cs)) :: lex csX
                   else lexId (hd csX ::cs) (tl csX)
```

Aufgabe 6: Parsen

Wir betrachten im Folgenden die sogenannten *regulären Ausdrücke*.

- Es gibt drei verschiedene Operatoren: Kleene'scher Stern $*$, Konkatenation $\circ$ und Vereinigung $+$.
- $*$ klammert stärker als $\circ$ und $\circ$ stärker als $+$ (und somit klammert $*$ auch stärker als $+$)
- $+$ und $\circ$ klammern implizit links (wie die arithmetischen Operatoren auch).
- $*$ ist ein unärer Operator, d.h. ist φ ein regulärer Ausdruck, so ist φ^* auch ein regulärer Ausdruck.
- $+$ und $\circ$ sind binäre Operatoren, d.h. sind φ und ψ reguläre Ausdrücke, so sind es auch $\varphi + \psi$ und $\varphi \circ \psi$.

Wortfolgen von regulären Ausdrücken werden mit den folgenden Token dargestellt:

```
datatype token = STERN | PLUS | KRINGEL | CON of string | LPAR | RPAR
```

Gültige reguläre Ausdrücke wären z.B. $A \circ B + B \circ A$, A^*, $A + B$ und $(A + B)^*$. Konstanten sind also beliebige Strings.

Hinweis: Sie müssen die Bedeutung von regulären Ausdrücke nicht kennen! Um einen Parser zu schreiben, brauchen Sie nur die Klammerungsstärke und den Aufbau der Ausdrücke.

Teilaufgabe 6.1 (*Links-rekursive Grammatiken*) **4 Punkte**
Stellen Sie eine links-rekursive Grammatik für reguläre Ausdrücke auf. Sie dürfen die Kategorie *con* verwenden, die für einen beliebigen String steht.

plusexp ::= [*plusexp* "+"] *kringelexp*
kringelexp ::= [*kringelexp* "∘"] *sternexp*
sternexp ::= *pexp* ["*"]
pexp ::= *con* | "(" *plusexp* ")"

wobei *con* einen beliebigen String bezeichnet.

Teilaufgabe 6.2 (*RA-taugliche Grammatik*) **6 Punkte**
Machen Sie Ihre Grammatik aus Aufgabenteil 6.1 RA-tauglich.

plusexp ::= *kringelexp plusexp'*
plusexp' ::= ["+" *kringelexp plusexp'*]
kringelexp ::= *sternexp kringelexp'*
kringelexp' ::= ["∘" *sternexp kringelexp'*]
sternexp ::= *pexp* ["*"]
pexp ::= *id* | "(" *plusexp* ")"

wobei *id* einen beliebigen String bezeichnet.

Teilaufgabe 6.3 **5 Punkte**

Schreiben Sie einen Parser zu Ihrer Grammatik aus 6.2. Dabei sollen Sie folgenden Datentypen *exp* verwenden:

```
datatype exp = Stern of exp | Plus of exp * exp | Kringel of exp * exp | Con of string
```

```
fun plusexp ts = plusexp' (kringelexp ts)
and plusexp' (e, PLUS::tr) = plusexp' (extend (e,tr) kringelexp Plus)
  | plusexp' s = s
and kringelexp ts = kringelexp' (sternexp ts)
and kringelexp' (e, KRINGEL::tr) = kringelexp' (extend (e,tr) sternexp Kringel)
  | kringelexp' s = s
and sternexp ts = case pexp ts of
    (a, STERN::tr) => (Stern(a),tr)
    | s => s
and pexp (CON z :: tr) = (Con z, tr)
  | pexp (LPAR :: tr) = match (plusexp tr) RPAR
  | pexp _  = raise Error "pexp"
```

Alternative Deklarationen ohne *match* und *extend*:

```
...
and plusexp' (e, PLUS::tr) = plusexp' (case (kringelexp tr) of (e', tr') => (Plus (e, e')), tr')
  | plusexp' s = s
...
and pexp (CON z :: tr) = (Con z, tr)
  | pexp (LPAR :: tr) = case (plusexp tr) of (a, RPAR :: tr') => (a, tr')
                                           | s => raise Error "match"
  | pexp _  = raise Error "pexp"
```

Aufgabe 7: Datenstrukturen

Teilaufgabe 7.1 (*Hash-Map — abstrakt*) **10 Punkte**
In dieser Aufgabe betrachten wir eine Hash-Map.

Eine Map ist eine Datenstruktur, die einer funktionalen Relation $R \subseteq X \times Y$ entspricht. Jedem $x \in X$ (Schlüssel) wird höchstens ein $y \in Y$ (Wert) zugeordnet. In dieser Aufgabe betrachten wir nur einen Teil aller möglichen Operationen, die man für eine Map implementieren kann. Wir wollen für einen Schlüssel x den zugehörigen Wert y abfragen können, zu einem noch nicht vorhandenen Schlüssel x den Wert y einfügen können und eine Map aus zwei gleich langen Listen, eine für die Schlüssel und eine für die Werte, erstellen können.

Eine mögliche Art eine Map zu implementieren ist die sogenannte Hash-Map. Wir erlauben eine sehr große Anzahl an Schlüsseln, daher ist es nicht so einfach eine Map so zu konstruieren, dass man zu einem Schlüssel schnell den zugehörigen Wert findet. Die Hash-Map zählt zu den schnellsten Arten der Implementierung. Für die Funktionalität einer Hash-Map ist eine Hash-Prozedur nötig, die einem beliebigen Schlüssel x einen Wert $z \in Z$ (Hash) zuordnet. Der Hash z ist eine beliebige ganze Zahl, muss aber für ein bestimmtes x immer gleich bleiben.

In unserer Hash-Map arbeiten wir intern mit einem Vektor der Größe 10, der aus einer Liste von $X * Y$-Paaren besteht. Wenn wir einen Wert y für Schlüssel x und $hash\ x = z$ in unserer Map speichern wollen, fügen wir das Paar (x, y) zur Liste des i-ten Eintrags des Vektors hinzu, wobei i der Rest der Division von z durch 10 ist. Wenn wir nach dem Wert y fragen, suchen wir wieder die i-te Liste im Vektor (i wie oben) und laufen so lange über die Liste, bis wir das Paar mit dem Schlüssel x gefunden haben und geben y zurück. Wir verwenden α *option* für das Finden von Werten, sodass wir einfach $NONE$ zurückgeben, wenn Schlüssel x nicht gefunden wird.

Ergänzen Sie die folgende Datenstruktur für den Sonderfall, in dem wir für die Menge X die Menge aller Strings nutzen. Beachten Sie die Vektor-Prozeduren in der Liste der erlaubten Prozeduren.

```
signature HASHMAP = sig
          type 'a hashMap
          val getValue: 'a hashMap -> string -> 'a option
          val addValue: 'a hashMap -> string -> 'a -> 'a hashMap
          val hashMap: string list -> 'a list -> 'a hashMap
end

structure HashMap :> HASHMAP = struct
          type 'a hashMap = ((string*'a) list) vector

          fun hash s = foldl (fn (c, a) => a + ord c) 0 (explode s)

          fun getValue m s = let
                  val i = (hash s mod 10)



                  in









                  end
```

```
        fun keyExists m s = case getValue m s of
                NONE => false
                | SOME _ => true

        fun addValue m s v = let
                val i = (hash s mod 10)



        in
                if keyExists m s
                then raise Subscript  (* Schluessel wurde bereits verwendet *)
                else




        end

        fun fillMap m nil nil = m   (* Hilfsprozedur fuer Prozedur hashMap *)
        | fillMap m (s::sr) (v::vr) =



        | fillMap _ _ _ = raise Subscript   (* Die Listen waren unterschiedlich lang *)

        fun hashMap sl al = fillMap (Vector.tabulate (10, (fn _ => nil))) sl al

end



signature HASHMAP = sig
        type 'a hashMap
        val getValue: 'a hashMap -> string -> 'a option
        val addValue: 'a hashMap -> string -> 'a -> 'a hashMap
        val hashMap: string list -> 'a list -> 'a hashMap
end

structure HashMap :> HASHMAP = struct
        type 'a hashMap = ((string*'a) list) vector

        fun hash s = foldl (fn (c, a) => a + ord c) 0 (explode s)

        fun getValue m s = let
                val i = (hash s mod 10)
                val items = Vector.sub (m, i)
        in
                case List.filter (fn (is, iv) => is = s) items of
                        [(is, iv)] => SOME iv
                        | nil => NONE
                        | _ => raise Subscript  (* > 1 items with s exist *)
        end
```

```
        fun keyExists m s = case getValue m s of
                NONE => false
                | SOME _ => true

        fun addValue m s v = let
                val i = (hash s mod 10)
                val items = Vector.sub (m, i)
        in
                if keyExists m s
                then raise Subscript  (* item with key s already exists *)
                else Vector.update (m, i, ((s,v)::items))
        end

        fun fillMap m nil nil = m
        | fillMap m (s::sr) (v::vr) = fillMap (addValue m s v) sr vr
        | fillMap _ _ _ = raise Subscript        (* different lengths *)

        fun hashMap sl al = fillMap (Vector.tabulate (10, (fn _ => nil))) sl al

end
```

Teilaufgabe 7.2 (*Hash-Map — konkret*) **2 + 1 Punkte**

In dieser Aufgabe sollen Sie die Datenstruktur aus der vorigen Aufgabe benutze.

(a) Erstellen Sie eine Hash-Map, die **direkt nach der Erstellung** nur Ihren Namen auf Ihre Matrikelnummer abbildet.

```
val map = HashMap.hashMap ["Your Name"] [1234567]
```

(b) Fügen Sie der Map einen Eintrag hinzu, der den Namen „Guybrush Threepwood“ (ein mächtiger Pirat) an 2442424 bindet.

```
val map = HashMap.addValue map "Guybrush Threepwood" 2442424
```

Aufgabe 8: Multiple Choice

Betrachten Sie diese Deklarationen.

```
val a = ref 5
val b = let val r = ref 0 in (fn () => (r := !r + 1; !r))
val c = fn () => let val r = ref 0 in (fn () => (r := !r + 1; !r))
val d = c ()
val e = c ()
val f = d
```

Welche Aussagen sind wahr?

Warnung: Die Aufgaben sind unabhängig voneinander, jede Aussage ist *direkt* nach dem Ausführen der obigen Deklarationen zu bewerten.

Falsche Antworten geben Punktabzug.

Teilaufgabe 8.1 **$10 \cdot \frac{1}{2} = 5$ Punkte**

(a)	Ob $!(ref\ 1) = !(ref\ 2)$ zu $true$ oder $false$ auswertet, hängt vom Zufall ab.	wahr ☐	falsch ■
(b)	$ref\ 0 = ref\ 0$ wertet zu $true$ aus	wahr ☐	falsch ■
(c)	$!(ref\ 0) = !(ref\ 0)$ wertet zu $false$ aus	wahr ☐	falsch ■
(d)	$d\,() = e\,()$ wertet zu $true$ aus	wahr ■	falsch ☐
(e)	$d\,() = f\,()$ wertet zu $true$ aus	wahr ☐	falsch ■
(f)	$d\,() = f\,() - 1$ wertet zu $true$ aus	wahr ■	falsch ☐
(g)	$(d\,(); d\,(); d\,()) = f\,()$ wertet zu $true$ aus	wahr ☐	falsch ■
(h)	Listen ermöglichen einen Zugriff auf Elemente in konstanter Laufzeit.	wahr ☐	falsch ■
(i)	Alle Werte, die den Typ $int \to (bool * int\ ref\ list)$ haben, sind imperativ.	wahr ☐	falsch ■
(j)	Alle Werte, die den Typ $int\ list\ ref$ haben, sind imperativ.	wahr ■	falsch ☐

Hinweise:

(a) Beim uns bekannten Fragment von SML hängen Ergebnisse nie von Zufall ab.

(e) d und f verwenden den gleichen Zähler, also werten beide Aufrufe (die nacheinander passieren) zu verschiedenen Werten aus. $d\,() = d\,()$ würde aus dem gleichen Grund zu $false$ auswerten.

(f) Hier wird die rechte Seite um 1 verringert, was den Effekt vom letzten Aufgabenteil ausgleicht.

(h) Zugriff in konstanter Laufzeit wird von Vektoren ermöglicht.

(i) Die Abstraktion $fn\ x \Rightarrow (true, nil)$ hat den geforderten Typ und ist nicht imperativ.

(j) Achtung, Verwechslungsgefahr: Bei $int\ ref\ list$ wären nicht alle Werte imperativ.

Aufgabe 9: Stapelmaschinen und Übersetzer (Zusatz)

Hinweis: Da die Aufgaben zu Kapitel 16 in den Übungen noch nicht hinreichend besprochen wurden, ist dies nur eine Zusatzaufgabe. Für die echte Klausur ist der in Vorlesung und Übungsblättern behandelte Stoff relevant.

Hinweis: *Die Definition einer Stapelmaschine finden Sie auf Seite 3.*

Teilaufgabe 9.1 (*Stapelmaschine*) **4 Punkte**
Übersetzen Sie das folgende W-Programm in M:

```
var n
var res:=1
while 1<=n do
  res:= res*n
  n:=n-1
end
```

Das hier angegebene Programm geht davon aus, dass das Argument n schon auf dem Stack liegt. Die Zeile **var n** wird nicht übersetzt.

```
[(con 1),                              (* res := 1 *)
 (getS 0),(con 1), leq, (cbranch 10), (* while 1<=n do *)
 (getS 1), (getS 0), mul, (putS 1),    (* res:=res*n *)
 (con 1), (getS 0), sub, (putS 0),     (* n:=n-1 *)
 (branch ~12),
 halt]
```

Teilaufgabe 9.2 (*Aufrufrahmen*) **3 Punkte**
Erklären Sie *kurz*, was ein Aufrufrahmen ist.

Der Aufrufrahmen enthält die Rücksprungadresse, die lokalen Variablen und die Argumente eines Prozeduraufrufs.

Teilaufgabe 9.3 (*Rätsel*) **3 Punkte**
Was tut das folgende Programm? (Das Programm erwartet, dass zwei Argumente auf dem Stack liegen, bevor es ausgeführt wird.)

```
[(getS 0), (con 1), (getS 1), leq, (cbranch 2),(branch 8),
 (getS 0), mul, (con 1), (getS 1), sub, (putS 1), (branch ~11),  halt]
```

Das Programm berechnet x^n, wobei x das erste Argument und n das zweite Argument auf dem Stack ist.