

2. Probeklausur – Special Directors' Cut Edition™ Programmierung 1

Wintersemester 2012/13

Das Team der Tutoren

02. Februar 2013

Name

Matrikelnummer

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

Prüfen Sie dann zuerst, ob Sie alle **19** 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 Klausurssaals 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:

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

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.

Teilaufgabe 1.2

4 Punkte

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

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.

Teilaufgabe 1.4**5 + 1 Punkte**

Betrachten Sie die Prozedur:

$$\begin{aligned} p : \mathbb{N} &\rightarrow \mathbb{N} \\ p\ 0 &= 0 \\ p\ 1 &= 1 \\ p\ n &= p(n-1) + 1 + p(n-1) \quad n > 1 \end{aligned}$$

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 $n \in \mathbb{N} : p\ n = 2^n - 1$

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

Aufgabe 2: Induktion

Teilaufgabe 2.1 (...natürlich)

5 Punkte

Beweisen Sie die folgende Behauptung durch natürliche Induktion:

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

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

Teilaufgabe 2.2 (...verstärkt)

3 Punkte

Betrachten Sie folgende Deklaration einer Prozedur $count : int\ list \rightarrow int$, 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.

Teilaufgabe 2.3 (*...strukturell*)

0 Punkte

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

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

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

(b) $p : \mathbb{N} \rightarrow \mathbb{N}$
 $p\ 0 = 1$
 $p\ n = p(n - 1) \quad n > 0$

(c) $q : \mathbb{Z} \rightarrow \mathbb{N}$
 $q\ 0 = 1$
 $q\ n = q(n - 1) + p\ n \quad n > 0$
 $q\ n = q(n + 1) + p(-n) \quad n < 0$

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

(e) $h : \mathbb{N} \rightarrow \mathbb{N}$
 $h\ 0 = 9$
 $h\ 1 = 90$
 $h\ 2 = 42$
 $h\ 3 = 0$
 $h\ 4 = 234624562389897$
 $h\ n = h\left(\lfloor \frac{n}{5} \rfloor\right) + 789 \quad n \geq 5$

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) $@ : \mathcal{L}(\mathcal{X}) \times \mathcal{L}(\mathcal{X}) \rightarrow \mathcal{L}(\mathcal{X})$

$$nil@ys = ys$$

$$(x :: xr)@ys = x :: (xr@ys)$$

Größenfunktionen:

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

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

(b) $sauschwer : \mathbb{N} \rightarrow \mathbb{N}$

$$sauschwer\ 0 = 78$$

$$sauschwer\ n = sauschwer\ (n - 1) \quad n > 0$$

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

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

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
```

```
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)
```

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

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 erlauben 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*

$$\mathbf{Snil} \quad \frac{}{T \vdash nil :}$$

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

$$\mathbf{Sconsn} \quad \frac{T \vdash e_2 : \alpha \text{ list}}{T \vdash e_1 :: e_2 :} \qquad \mathbf{Scons} \quad \frac{T \vdash e_2 : t \text{ list}}{T \vdash e_1 :: e_2 :}$$

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

$$\mathbf{Soaln} \quad \frac{T \vdash e_2 : \alpha \text{ list}}{T \vdash e_1 @ e_2 :} \qquad \mathbf{Snoal} \quad \frac{T \vdash e_1 : \alpha \text{ list}}{T \vdash e_1 @ e_2 :}$$

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

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)) =

```

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:

$$\mathbf{Dnil} \frac{}{V \vdash nil \triangleright ()} \quad \mathbf{Dcons} \frac{V \vdash e_1 \triangleright v \quad V \vdash e_2 \triangleright v'}{V \vdash e_1 :: e_2 \triangleright (v, v')}$$

$$\mathbf{Dnoal} \frac{V \vdash e_1 \triangleright () \quad V \vdash e_2 \triangleright v}{V \vdash e_1 @ e_2 \triangleright v} \quad \mathbf{Doal} \frac{V \vdash e_1 \triangleright (v, v') \quad V \vdash e_2 \triangleright v'' \quad z = (v, v' @ v'')}{V \vdash e_1 @ e_2 \triangleright z}$$

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)) =
```

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 v 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.

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 Ausdrücken 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.

Teilaufgabe 6.2 (*RA-taugliche Grammatik*)

6 Punkte

Machen Sie Ihre Grammatik aus Aufgabenteil 6.1 RA-tauglich.

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
```

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 Schlüssel, 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 *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

```

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

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

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

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

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 <i>true</i> oder <i>false</i> ausgewertet, hängt vom Zufall ab. | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (b) $ref\ 0 = ref\ 0$ wertet zu <i>true</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (c) $!(ref\ 0) = !(ref\ 0)$ wertet zu <i>false</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (d) $d\ () = e\ ()$ wertet zu <i>true</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (e) $d\ () = f\ ()$ wertet zu <i>true</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (f) $d\ () = f\ () - 1$ wertet zu <i>true</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (g) $(d\ (); d\ (); d\ ()) = f\ ()$ wertet zu <i>true</i> aus | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (h) Listen ermöglichen einen Zugriff auf Elemente in konstanter Laufzeit. | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (i) Alle Werte, die den Typ $int \rightarrow (bool * int\ ref\ list)$ haben, sind imperativ. | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |
| (j) Alle Werte, die den Typ $int\ list\ ref$ haben, sind imperativ. | wahr <input type="checkbox"/> | falsch <input type="checkbox"/> |

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
```

Teilaufgabe 9.2 (Aufrufrahmen)

3 Punkte

Erklären Sie *kurz*, was ein Aufrufrahmen ist.

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]
```