2. Probeklausur zur Vorlesung Programmierung 1 im Wintersemester 2011/2012

Das Team der Tutoren

04. Februar 2012

Name

Matrikelnummer

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

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 100 Minuten zur Verfügung. Insgesamt können 100 Punkte erreicht werden. Zum Bestehen der Klausur genügen 50 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 vom Ablauf und dem Aussehen der Endklausur in Programmierung 1 zu geben. Dennoch ist sie insbesondere was Stoffauswahl und erlaubte Hilfsmittel angeht nicht repräsentativ für die Endklausur.

Falls Sie einen Aufgabenteil *nicht* lösen können, dürfen Sie die angegebenen Prozeduren und ihre Funktionalität dennoch in den darauffolgenden Aufgabenteilen zur Lösung verwenden.

Sie dürfen zur Lösung der Aufgaben keine nicht deklarierten Hilfsprozeduren verwenden. Während der Probeklausur dürfen Sie die linken Seiten der Klausur zum Schreiben benutzen.

1	2	3	4	5	6	7
10	15	10	10	20	25	10

Summe
90

Note

Aufgabe 1. (2 + 6 + 2 = 10 Punkte)

Die Datenstruktur *Schlange* kann, wie aus der Vorlesung bekannt, effizient durch die geschickte Verwendung zweier Listen implementiert werden.

a) Geben Sie die beiden Darstellungsinvarianten dieser effizienten Implementierung an.

b) Vervollständigen Sie nun die Implementierung:

```
structure QUEUE = struct
  type 'a queue = 'a list * 'a list
  val empty = ...

  fun snoc  ...

  fun head ...

  fun tail ...

end
```

Hinweis: Sie dürfen die Prozedur $rev : \alpha\ list \to \alpha\ list$ zum Reversieren einer Liste ohne vorherige Deklaration benutzen.

c) Erklären Sie kurz, weshalb es sinnvoll ist, akkumulierte Laufzeiten von Datenstrukturen zu betrachten.

Erklären Sie nun, wieso alle Operationen obiger Implementierung *akkumuliert betrachtet* konstante Laufzeit haben.

Aufgabe 2. (3 + 3 + 8 + 1 = 15 Punkte)

Sei folgende Prozedur gegeben:

$null : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$null(m, n) = \text{if } m \leq n \text{ then } 0 \text{ else } null(m, n + 2 \cdot (m + n) \text{ div } 3 + 1) +$
$null(n + 2 \cdot (m + n) \text{ div } 3, n + (m + n) \text{ div } 3 + 1) + null(n + (m + n) \text{ div } 3, n)$

a) Geben Sie Rekursionsfunktion und die Rekursionsrelation an.

b) Geben Sie eine natürliche Terminierungsfunktion an.

c) Zeigen Sie, dass *null* die Nullfunktion berechnet. Verwenden Sie dazu den Korrektheitssatz.

d) Bestimmen Sie die Komplexität der Ergebnisfunktion.

Aufgabe 3. (10 Punkte)

Sei

$F_0 := 0$

$F_1 := 1$

$F_n := F_{n-1} + F_{n-2} \quad \text{für} \quad n > 1$

Beweisen Sie per Induktion: $\forall n \in \mathbb{N}_+ : F_1^2 + F_2^2 + \cdots + F_n^2 = F_n \cdot F_{n+1}$. Begründen Sie jeden Beweisschritt.

Aufgabe 4. (6 + 4 = 10 Punkte)

Betrachten Sie folgende Prozedur:

$$
\begin{array}{ll}
\mathit{change} & : \mathcal{L}(\mathbb{N}) \to \mathcal{L}(\mathbb{N}) \\
\mathit{change}\ \mathit{nil} & = \mathit{nil} \\
\mathit{change}\ (x :: xs) & = (2 \cdot x) :: \mathit{change}\ xs
\end{array}
$$

a) Bestimmen Sie die rekursive Darstellung der Laufzeitfunktion von *change* bezüglich folgendener Größenfunktionen:

- $\lambda xs.\, |xs|$
- $\lambda xs.\, |xs| + 7$
- $\lambda xs. 2^{|xs|}$

b) Bestimmen Sie mit Hilfe der Rekurrenzsätze die Komplexität der jeweiligen Laufzeitfunktionen. Begründen Sie jeweils kurz, warum Sie den entsprechenden Rekurrenzsatz anwenden dürfen.

Aufgabe 5. ((2 + 3 + 3) + (2 + 5 + 5) = 20 Punkte)

a) Sei $A = \{1, 2, 3, 4, 5, 6\}$ eine Menge und sei $R = \{(1, 4), (1, 5), (4, 5), (5, 1)\}$ eine Relation über A.

(i) Zeichnen Sie den Graph der Relation.

(ii) Ist R funktional? Injektiv? Surjektiv auf A? Total auf A? Begründen Sie Ihre Antworten!

(iii) Geben Sie $R \circ R$ an! Was können Sie mit R und $R \circ R$ für die Transitivität von R folgern?

b) Es sei eine abstrakte Datenstruktur *RelSet* gegeben, die die üblichen Operationen auf Mengen über Paaren des Typs *int * int* spezifiziert. Zudem besitzt sie eine Operation *toList: set → (int * int) list*, die eine Menge wieder in eine Tupelliste umwandelt. Ihre Signatur sei dabei

type set
val emptyset : unit → set
*val insert : (int * int) list → set → set*
*val toList : set → (int * int) list*
*val isElement : (int * int) → set → bool*
val isSubset : set → set → bool
val union : set → set → set

Wir können nun Relationen als Elemente vom Typ *set* darstellen.

(i) Geben Sie einen Ausdruck vom Typ *set* an, der die Relation R darstellt.

(ii) Schreiben Sie eine Prozedur *closure: set → set*, die den transitiven Abschluss einer Relation bestimmt, d.h., die in eine gegebene Relation *genau* die Kanten einfügt, die benötigt werden, um die Relation transitiv zu machen.

Tipp: Schreiben Sie zunächst eine Prozedur *compose: set → set → set*, die die Komposition zweier binärer Relationen berechnet. Erinnern Sie sich dann an die mathematische Definition von Transitivtät aus dem Skript (bzw. Teilaufgabe *a)*(iii)!).

(iii) Sie haben in der Vorlesung verschiedene Möglichkeiten kennengelernt, die abstrakte Datenstruktur *Mengen* zu implementieren. Diese hatten für die jeweiligen Mengenoperationen unterschiedliche Laufzeiten. Welche Implementierung würden Sie für `RelSet` wählen, damit Ihre Prozedur `closure` eine möglichst geringe Laufzeitkomplexität hat. Vergessen Sie nicht, Nebenkosten zu betrachten. Gehen Sie dabei davon aus, dass *toList* stets mit linearer Komplexität realisiert ist. Begründen Sie Ihre Entscheidung!

Aufgabe 6. (5 + 5 + 5 + 5 + 5 = 25 Punkte)

Es sei die folgende abstrakte Grammatik für logische Ausdrücke gegeben.

$$x \in Id = \mathbb{N}$$
$$\phi, \psi \in exp = true \mid x \mid \bot x : \phi \mid \neg\phi \mid \phi \to \psi$$

Das Ziel ist es einen Interpreter für logische Ausdrücke zu schreiben, d.h., der einen logischen Ausdruck zu Werten aus der Menge $\{0, 1\}$ auswertet. Die Konstant *true* sowie die Operatoren $\neg$ und $\to$ sollten Ihnen aus der Aussagenlogik bereits bekannt sein. Weiter steht der Ausdruck x für eine logische Variable. Der Ausdruck $\bot x : \phi$ soll wahr werden, wenn ϕ bezüglich x eine Tautologie ist, d.h. ϕ zu 1 auswertet, egal ob die Variable x den Wert 1 oder den Wert 0 repräsentiert.

Z.B. soll $\bot x : \bot y : \neg(x \to x) \to y$ zu 1 auswerten, der Ausdruck $\bot x : x$ jedoch zu 0.

Eine Variable x heißt gebunden in ϕ', wenn sie nur in Teilausdrücken der Form $\bot x : \phi$ vorkommt.

a) Geben Sie Typen und Typinferenzregeln an, die einem Ausdruck nur dann unter der leeren Umgebung einen zulässigen Typen zuweisen, falls alle Variablen gebunden sind.

b) Schreiben Sie nun die Prozedur *elab*, die den Typ eines Ausdrucks bestimmt, falls dieser existiert, und ansonsten eine Ausnahme wirft. Deklarieren Sie dazu selbstständig alle notwendigen Datentypen (z.B. unter anderem für Ausdrücke) und Ausnahmen.

c) Geben Sie nun die Inferenzregeln für die dynamische Semantik an.

d) Schreiben Sie eine Prozedur *eval*, die einen aussagenlogischen Ausdruck auswertet. Deklarieren Sie dazu alle notwendigen Datentypen und Ausnahmen (sofern nicht schon bei Teil *b)* geschehen). Die Auswertung muss nur für wohlgetypte Ausdrücke ein korrektes Ergebnis liefern.

e) *Zusatz:* Schreiben Sie einen Lexer *lex: char list* $\rightarrow$ *token list*. Sie dürfen für die Operatoren die oben definierte mathematische Schreibweise und folgenden Datentyp token verwenden.

```
datatype token = NOT |  VAR of var | TAUTO | TRUE |  ARROW | LPAR |
RPAR | DOT
```

f) Stellen Sie eine konkrete Grammatik für logische Ausdrücke auf. Wandeln Sie diese dann in eine RA-taugliche Grammatik um, die die Ihnen aus der Vorlesung bekannten Klammerregeln für logische Ausdrücke umsetzt.

Hinweis: Sie dürfen die beiden Schritte gerne in einem Schritt kombinieren und direkt eine RA-taugliche konkrete Grammatik angeben!

g) *Zusatz:* Schreiben Sie zu Ihrer Grammatik einen Parser.

Aufgabe 7. (6 + 4 = 10 Punkte)

Das Ihnen hoffentlich noch bekannte Restaurant *Hermanns & Eisentraut* braucht erneut Ihre Hilfe. Dieses Mal soll Zeit beim Würzen der Gerichte gespart werden.

Sie empfehlen die Kochmaschine vom Typ *Kochomat 5000*, die zum Glück auf SML-Basis arbeitet und von Ihnen ein Tupel des Typs *wuerze* als Eingabe verlangt:

```
type wuerze = (unit -> bool) * (unit -> bool)
```

Dabei steht die erste Komponente für *Salz* und die zweite für *Pfeffer*. Die Maschine geht nun wie folgt vor: Sie testet für jede der beiden Tupelkomponenten einzeln, ob die jeweilige Prozedur *true* ausgibt. Wenn ja, so würzt sie das Gericht. Und das Ganze jeweils so lange, bis die Prozedur *false* ausgibt.

a) Schreiben Sie nun einen Generator $gen : int * int \rightarrow wuerze$, welcher zu der Eingabe (s, p) das passende Tupel für die Kochmaschine zurückgibt, sodass diese s-mal mit Salz und p-mal mit Pfeffer würzt.

b) Schreiben Sie die Prozeduren $countSalz : wuerze \rightarrow int$ und $countPfeffer : wuerze \rightarrow int$, welche den Würzegrad für Salz beziehungsweise Pfeffer bestimmen. Der Würzegrad ist die Anzahl der Würzvorgänge für die jeweilige Zutat.

Hinweis: Dabei ist der Zustand der Argumentprozeduren nach einem Aufruf von *countSalz* oder *countPfeffer* egal.