Name: [ ] Vorname: [ ] Matr.–Nr.: [ ]

Technische Universität München
Fakultät für Informatik
Prof. Dr. H. Seidl

SoS 2005
16. Juli 2005

Klausur zu Einführung in die Informatik II

Hinweis: In dieser Klausur können Sie insgesamt 60 Punkte erreichen. Zum Bestehen benötigen Sie mindestens 24 Punkte.

Aufgabe 1 Java-GUI (10 Punkte)

Programmieren Sie ein Applet, dessen Hintergrundfarbe rot oder blau ist. Das Applet soll einen Knopf mit der Aufschrift `Change Colour` haben. Wird dieser Knopf betätigt, wechselt die Hintergrundfarbe von rot nach blau bzw. umgekehrt.

Hinweis: Zum Setzen der Hintergrundfarbe stellt die Klasse `Applet` die Methode `setBackground(Color c)` bereit.

Aufgabe 2 Bäume (1 + 3 + 3 + 3 = 10 Punkte)

In dieser Aufgabe sollen Sie mithilfe **rein funktionaler** Konstrukte eine Datenstruktur zur Repäsentation von Bäumen entwickeln.

a) Definieren Sie einen OCaml-Typen `'a tree` zur Repräsentation von Bäumen, in dem jeder Knoten beliebig viele Kindknoten besitzen kann. In jedem Knoten eines solchen Baumes soll dabei eine Information vom Typen `'a` gespeichert werden.

b) Definieren Sie eine OCaml-Funktion `size : 'a tree -> int`, die – für einen als Argument übergebenen Baum – die Anzahl der im Baum enthaltenen Knoten bestimmt.

c) Definieren Sie eine OCaml-Funktion `map : ('a -> 'b) -> 'a tree -> 'b tree`, die einen Baum vom Typen `'b tree` aus dem als Argument übergebenen Baum berechnet. Für einen Aufruf `map f t` soll `map` einen Baum `s` berechnen, der strukturgleich zum Argumentbaum `t` ist. D.h. die beiden Bäume sollen sich lediglich durch die Knoten-Informationen unterscheiden. Eine Knoten-Information für `s` erhält man, indem man die Funktion `f : 'a -> 'b` auf eine Knoten-Information von `t` anwendet.

d) Definieren Sie eine OCaml-Funktion `to_list : 'a tree -> 'a list`, die eine Liste aller Knoten-Informationen des als Argument übergebenen Baumes berechnet.

Aufgabe 3 Graphen (2 + 4 + 4 = 10 Punkte)

Gegeben sei der wie folgt definierte Daten-Typ `graph` zur Repräsentation gerichteter Graphen

```
type node = int
type graph = node list array
```

Bemerkung: Der hier angegebene Daten-Typ entspricht dem in der Vorlesung angegebenen Daten-Typ `'a graph`, bis auf die Tatsache, dass die Kanten hier nicht mit einer Information versehen werden können.

a) Definieren Sie eine Funktion `add_edge : graph -> node -> node -> unit` zum Hinzufügen einer Kante zu einem gerichteten Graphen.

b) Definieren Sie eine Funktion `count : graph -> node -> int`, die – zu einem gerichteten Graphen `g` und einem Knoten `s` – die Anzahl der vom Knoten `s` erreichbaren Knoten berechnet. Dabei soll der Knoten `s` mitgezählt werden.

c) Definieren Sie eine OCaml-Funktion `make_bidirected` die als Argument einen gerichteten Graphen erhält und dann einen Graphen konstruiert und zurückliefert, der dem übergebenen Graphen entspricht und zusätzlich zu jeder Kante auch eine Gegenkante enthält.

Aufgabe 4 Greedy-Algorithmen (10 Punkte)

Gegeben sei der in der vorherigen Aufgabe angegebene Daten-Typ zur Respräsentation gerichteter Graphen.

Für einen **ungerichteten** Graphen $G = (V,E)$ heißt eine Knoten-Menge $V' \subseteq V$ *Independent-Set*, falls für je zwei verschiedene Knoten v_1 und v_2 aus V' keine Kante $\{v_1, v_2\}$ im Graphen G existiert. Abbildung 1 zeigt ein *Independent-Set*.

Implementieren Sie eine OCaml-Funktion, die ein möglichst großes *Independent-Set* eines Graphen berechnet. Gehen Sie dabei davon aus, dass der ungerichtete Graph durch einen gerichteten Graphen repräsentiert wird, bei dem es zu jeder Kante auch eine Gegenkante gibt.

Hinweis: Benutzen Sie eine Greedy-Strategie, die sukzessive neue Knoten zum bisherigen Ergebnis hinzufügt.

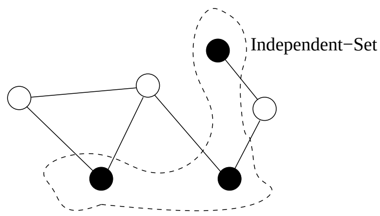


Abbildung 1: Ein Independent-Set

Aufgabe 5 Verifikation funktionaler Programme (10 Punkte)

Die Funktion `length` ist folgendermaßen gegeben:

```
let rec length = fun ls -> match ls with
    | [] -> 0
    | x :: xs -> 1 + length xs
```

Die Funktion `app` sei wie in der Vorlesung definiert:

```
let rec app = fun x -> fun y ->
  match x with
      [] -> y
    | x::xs -> x :: app xs y
```

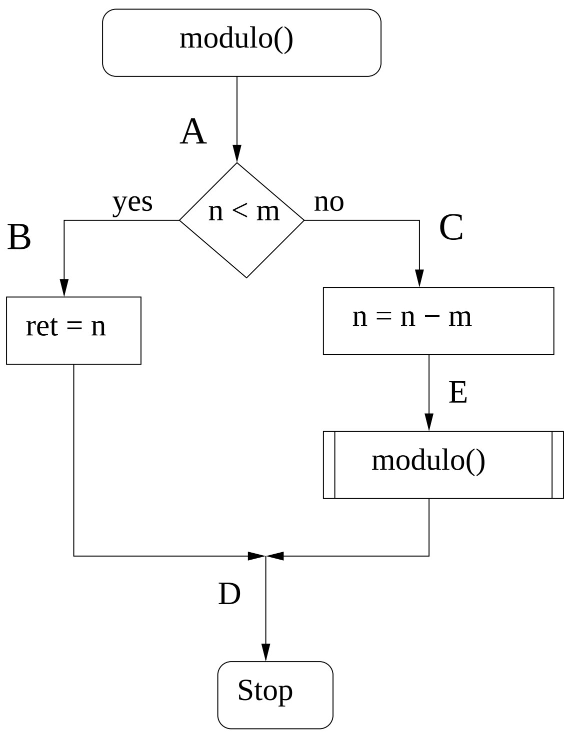
Zeigen Sie, dass für alle Listen *ls* und *ks* gilt:

$$\texttt{length}\,(\texttt{app ls ks}) = \texttt{length ls} + \texttt{length ks}$$

Aufgabe 6 Verifikation eines Mini-Java-Programms (10 Punkte)

Die rekursive Prozedur `modulo` zur Berechnung des Restes bei Division zweier positiver Zahlen `m` und `n` verwendet ausschließlich die globalen Variablen `ret`, `n` und `m`. Die Variable `ret` dient hierbei zur Rückgabe des Ergebnisses. Die Prozedur wird durch nebenstehenden Kontrollflussgraphen beschrieben.

(**Hinweis:** Die hier gefragten Annotationen sind in nebenstehendem Graphen mit den Buchstaben `A` - `E` gekennzeichnet. Verwenden Sie bei Ihrer Lösung diese Bezeichnungen.)



Verifizieren Sie die folgende globale Hypothese:

$$\{m > 0 \wedge n \geq 0 \wedge m = l_2 \wedge n = l_1\} \;\; \text{modulo()} \;\; \{0 \leq ret < l_2 \wedge \exists z \in \mathbb{Z} : (z \geq 0) \wedge (l_2 \cdot z + ret = l_1)\}$$

Geben Sie lokal konsistente Zusicherungen `A`-`E` an und zeigen Sie, dass diese lokal konsistent sind. Gegebenfalls durchzuführende Äquivalenzumformungen, Abschwächungen oder Verstärkungen sind zu erläutern!