15. Übungsblatt zu Programmierung WS 2002 / 03

Prof. Dr. Gert Smolka und Dipl.-Inform. Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws02/

Abgabe: Montag, 17. Februar 2003

Schicken Sie Ihre Lösungen per E-Mail an Ihren Übungsgruppenleiter. Geben Sie in der E-Mail zuerst ihren Namen und die Nummer Ihrer Übungsgruppe an. Bitte reichen Sie alle Lösungen mit nur einer E-Mail ein.

Aufgabe 15.1: Graphen ($16 = 4 * 4$) Seien die folgenden Graphen gegeben:

(a) $V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 5), (2, 1), (2, 3), (2, 4), (3, 5), (6, 2), (6, 3)\}$

(b) $V = \{1, 2, 3, 4, 5, 6, 7\}$
$E = \{(2, 7), (3, 1), (3, 6), (4, 2), (4, 3), (7, 5)\}$

(c) $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$E = \{(2, 1), (2, 6), (3, 8), (4, 4), (5, 2), (5, 7), (6, 1), (7, 6)\}$

(d) $V = \{1, 2, 3, 4, 5\}$
$E = \{(1, 4), (2, 4), (3, 2), (3, 5), (4, 3), (5, 1)\}$

Zeichnen Sie diese Graphen und beantworten Sie für jeden Graphen die folgenden Fragen:

- Welche Größe und welche Tiefe hat der Graph?
- Welche Quellen und welche Senken hat der Graph?
- Ist der Graph zyklisch? Wenn ja, geben Sie einen Zyklus an.
- Ist der Graph zusammenhängend? Stark zusammenhängend?
- Ist der Graph ein Wald? Ein Baum?

Aufgabe 15.2: Graphen mit Standard ML ($24 = 3 * 8$) Seien Graphen in Standard ML gemäß der Typdeklarationen

```
datatype 'a state = G of bool * 'a * 'a graph list
withtype 'a graph = 'a state ref
```

realisiert. Das Boolesche Feld fungiert als Merkfeld. Als initialer Wert wird `false` verwendet. Beispielgraphen können mit den folgenden Prozeduren aufgebaut werden:

```
fun cons x gs =  ref(G(false,x,gs))
val cons : 'a → 'a graph list → 'a graph

fun assign (g as ref(G(m,x,_))) gs = g:= G(m,x,gs)
val assign : 'a graph → 'a graph list → unit
```

(a) Schreiben Sie eine iterative Prozedur

```
val reset : 'a graph list → unit
```

die das Merkfeld aller von der Argumentliste aus erreichbaren Knoten auf `false` zurücksetzt. Nehmen Sie dabei an, dass `reset` nur dann angewendet wird, wenn von allen mit `false` markierten Knoten nur mit `false` markierte Knoten erreichbar sind.

(b) Schreiben Sie eine Prozedur

```
val tree : 'a graph → bool
```

die testet, ob ein Graph ein Baum ist. Verwenden Sie eine interative Hilfsprozedur

```
val tree' : 'a graph list → bool
```

die die Merkfelder der besuchten Knoten auf `true` setzt. Die Prozedur `tree` geht davon aus, dass alle Merkfelder initial auf `false` stehen und stellt diesen Zustand nach dem Aufruf von `tree'` wieder her.

(c) Schreiben Sie eine Prozedur

```
val count : 'a graph list → int
```

die die Anzahl der von der Argumentliste aus erreichbaren Knoten liefert. Verwenden Sie eine interative Hilfsprozedur

```
val count' : int → 'a graph list → int
```

die die Merkfelder der besuchten Knoten auf `true` setzt. Die Prozedur `count` geht davon aus, dass alle Merkfelder initial auf `false` stehen und stellt diesen Zustand nach dem Aufruf von `count'` wieder her.

Aufgabe 15.3: Listen mit VH $(24 = 3 * 8)$ Sie sollen Listen mit der virtuellen Maschine VH realisieren. Die leere Liste soll durch die Zahl 0 dargestellt werden. Eine nichtleere Liste soll durch eine Block in der Halde dargestellt werden, dessen erste Zelle den Kopf und dessen zweite Zelle den Rumpf der Liste verfügbar machen.

(a) Geben Sie Befehlssequenzen für die folgenden Prozeduren an:

```
cons : 'a * 'a list → 'a list
null : 'a list → bool
hd   : 'a list → 'a
tl   : 'a list → 'a list
```

Nehmen Sie dabei an, dass `hd` und `tl` nur auf nichtleere Listen angewendet werden.

(b) Geben Sie eine Befehlssequenz für eine Prozedur

```
gen : int → int list
```

an, die zu $n \in \mathbb{N}$ die Liste $[0, \ldots, n]$ liefert. Realisieren Sie `gen` mit einer iterativen Hilfsprozedur

```
gen' :  int list * int → int list
```

Achten Sie darauf, dass Sie nur Endaufrufe verwenden. Nehmen Sie an, dass Ihre Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird und schreiben Sie Ihre Befehlssequenz so, dass die Prozedur `gen` die Adresse 0 bekommt.

(c) Geben Sie eine Befehlssequenz für eine Prozedur

```
sum : int list → int
```

an, die die Summe der Elemente einer Liste liefert. Realisieren Sie `sum` mit einer iterativen Hilfsprozedur

```
sum' :  int * int list → int
```

Achten Sie darauf, dass Sie nur Endaufrufe verwenden. Nehmen Sie an, dass Ihre Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird und schreiben Sie Ihre Befehlssequenz so, dass die Prozedur `sum` die Adresse 0 bekommt.

Aufgabe 15.4: Speicher ($36 = 16 + 20$) Sie sollen einen Speicher realisieren, in dem Graphen abgelegt werden können, die aus terminalen Knoten und aus binär verzweigenden Knoten gebildet sind. Der Speicher soll mit der folgenden Signatur implementiert werden:

```
signature STORE =  sig
  exception Error
  eqtype value
  val fromInt : int   → value
  val toInt   : value → int
  val isInt   : value → bool
  val pair    : value * value → value
  val first   : value → value
  val second  : value → value
  val updateF : value * value → unit
  val updateS : value * value → unit
end
```

Die Signatur bezeichnet Knoten als Werte. Terminale Knoten sind ganze Zahlen und werden mit `fromInt` eingeführt. Verzweigende Knoten werden mit `pair` eingeführt und als imperative Paare bezeichnet. Mit den Prozeduren `updateF` und `updateS` können die Komponenten von imperativen Paaren auf andere Werte gesetzt werden.

Der Speicher soll analog zum Speicher der virtuellen Maschine VH realisiert werden (ohne Speicherbereinigung). Die Ausnahme `Error` soll geworfen werden, wenn ein Wert nicht den erwarteten Typ hat oder wenn eine Zahl zu groß ist (bei `fromInt`).

Die Blöcke für die Paare sollen 3 Zellen haben, wobei die erste Zelle wird als Merkzelle benutzt wird und mit -1 initialisiert wird.

Erweitern Sie Ihren Speicher um eine Operation

```
clone : value → value
```

die eine Kopie des von einem Knoten aus erreichbaren Graphen erstellt und den neuen Wurzelknoten liefert. Gehen Sie dabei analog zur Speicherbereinigung für VH vor und verwenden Sie zwei Hilfsprozeduren `copyBlock` und `clone'`. Zusätzlich benötigen Sie eine Hilfsprozedur `reset`, die die Merkzellen der Knoten des Ausgangsgraphens wieder auf -1 setzt, nachdem die Kopie erstellt ist. Der dafür benötigte Stapel soll in der Reihung des Speichers realisiert werden.

Verwenden Sie keine Rekursion und keine Listen.