13. Übungsblatt zu Programmierung (WS 1999/2000)

Andreas Rossberg, Prof. Gert Smolka

Abgabe: Freitag, 4. Februar 2000 vor der Vorlesung

Alle Signaturen finden Sie unter `http://www.ps.uni-sb.de/courses/info1-ws99/uebung/sml/blatt13/`.

Aufgabe 13.1: Schalter (4 Punkte)

(a) Schreiben Sie eine imperative Prozedur `toggle: unit -> bool`, die bei jeder Anwendung die Negation des vorherigen Ergebnisses liefert. Beispielsweise:

```
(toggle(), toggle(), toggle(), toggle())
(true,false,true,false) : bool * bool * bool * bool
```

(b) Schreiben Sie eine Prozedur

```
newToggle: unit -> unit -> bool
```

die bei jeder Anwendung eine neue Schalterprozedur wie in (a) liefert.

Aufgabe 13.2: Schleifen (8 Punkte)

Schreiben Sie nicht-rekrusive Prozeduren mit Hilfe von Schleifen wie folgt:

(a) Die Prozedur `sqrt: int -> int` berechnet für $n \in \mathbb{N}$ die ganzzahlige Quadratwurzel $\max\{k \in \mathbb{N} \mid k^2 \leq n\}$.

(b) Die Prozedur `length: 'a list -> int` berechnet die Länge einer Liste.

(c) Die Prozedur `fib: int -> int` berechnet für $n \in \mathbb{N}$ die die n-te Fibonacci-Zahl f_n.

Aufgabe 13.3: Schlangen als imperative Datenstruktur (4 Punkte)

Ähnlich wie Stapel können auch Schlangen als imperative Objekte verwirklicht werden. Betrachten Sie dazu die Signatur

```
signature IMP_QUEUE =
  sig
    type 'a queue
    exception Empty
    val queue:   unit -> 'a queue
    val put:     'a queue * 'a -> unit
    val get:     'a queue -> 'a                (* Empty *)
    val isEmpty: 'a queue -> bool
  end
```

Mit `queue` können neue Schlangen erzeugt werden. Mit `put` läßt sich ein Element an das Ende einer Schlange anfügen, mit `get` wird das erste Element aus der Schlange entfernt.

Schreiben Sie eine Struktur `ImpQueue`, die Schlangen als imperative Objekte gemäß obiger Signatur implementiert.

Aufgabe 13.4: Priorisierte Schlangen (8 Punkte)

Eine priorisierte Schlange (*priority queue*) ist ein Schlange, in der jedem Element eine Priorität zugeordnet ist. Die `get`-Operation liefert in einer priorisierten Schlange nicht einfach das älteste Element, sondern das älteste Element mit der höchsten Priorität. Neue Elemente sollen sich also vor ältere 'vordrängeln', deren Priorität niedriger ist.

Schreiben Sie eine Struktur `ImpPQueue`, die priorisierte Schlangen als imperative Objekte gemäß folgender Signatur implementiert:

```
signature IMP_PQUEUE =
  sig
    type 'a pqueue

    exception Empty

    val pqueue:  unit -> 'a pqueue
    val put:     'a pqueue * 'a * int -> unit
    val get:     'a pqueue -> 'a                    (* Empty *)
    val isEmpty: 'a pqueue -> bool
  end
```

Der Prozedur `put` wird neben dem einzufügenden Element auch dessen Priorität übergeben. Hohe Zahlen bedeuten hohe Prioritäten. Zum Beispiel:

```
open ImpPQueue
val q = pqueue() : string pqueue
put(q, "Horf", 7)
put(q, "Gismalda", 3)
put(q, "Pazo", 6)
(get q, get q, get q)
("Horf","Pazo","Gismalda") : string * string * string
```

Aufgabe 13.5: Reihungen (6 Punkte)

Schreiben Sie die folgenden Prozeduren:

(a) Eine Prozedur

```
foldlArray: ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
```

die entsprechend `foldl` für Listen die Elemente einer Reihung von links her faltet.

(b) Eine Prozedur

```
rotate: 'a array -> unit
```

die alle Elemente eines Arrays um eine Position nach rechts rotiert (das letzte Element wird dabei zum ersten Element).

Aufgabe 13.6: Binäre Suche (5 Punkte)

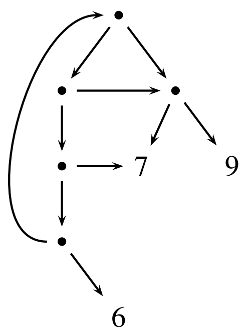
Schreiben Sie eine Prozedur

```
contains: int array * int -> bool
```

die für eine aufsteigend sortierte Reihung ermittelt, ob eine Zahl in der Reihung enthalten ist. Die Prozedur soll für Reihungen der Länge n die Laufzeit $O(\log n)$ haben. Zunächst wird dazu an Hand der mittleren Zelle der Reihung entschieden, in welcher Hälfte sich das Element gegebenenfalls befinden muss. Diese Idee wird rekursiv angewendet, bis das Element gefunden wurde, oder keine weitere Halbierung des Suchbereiches mehr möglich ist.

Aufgabe 13.7: Cons-Zellen (12 Punkte)

Cons-Zellen sind imperative Objekte, mit denen sich eine Klasse von Graphen darstellen lässt. Hier ist ein Beispiel:



Die Knoten dieser Graphen sind entweder ganze Zahlen, oder Cons-Zellen (mit • dargestellt). Von einer Cons-Zelle gehen immer genau zwei Kanten aus, die umgesetzt werden können.

(a) Schreiben Sie eine Struktur `Cons`, die Cons-Zellen gemäß folgender Signatur implementiert:

```
signature CONS =
  sig
    eqtype cons

    val number:    int -> cons
    val cons:      cons * cons -> cons

    val isNumber:  cons -> bool
    val value:     cons -> int             (* Domain *)
    val fst:       cons -> cons            (* Domain *)
    val snd:       cons -> cons            (* Domain *)

    val setFst:    cons * cons -> unit     (* Domain *)
    val setSnd:    cons * cons -> unit     (* Domain *)
  end
```

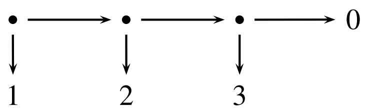
Dabei soll die Operation `cons` eine neue Cons-Zelle erzeugen, deren zwei Kanten auf die übergebenen Objekte zeigen. Die Operationen `fst` und `snd` liefern den erste bzw. zweiten Nachfolger einer Cons-Zelle. Die Prozeduren `setFst` und `setSnd` setzen die Nachfolger-Kanten einer Cons-Zelle um. Die Operation `value` liefert für die mit `number` n erzeugten Objekte die Zahl n.

Die Gleichheit auf dem Typ `cons` soll wie folgt definiert sein:

1. Jeder Aufruf der Operation `cons` liefert eine neue Cons-Zelle.
2. Die mit `number` erzeugten Objekte sind gleich genau dann, wenn sie die gleiche Zahl tragen.

(b) Schreiben Sie einen Ausdruck, der den oben gezeigten Graphen erzeugt und die oberste Cons-Zelle liefert.

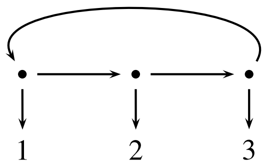
(c) Mit Hilfe von Cons-Zellen kann man Listen mit ganzzahligen Elementen repräsentieren. Die Liste `[1,2,3]` wird dabei wie folgt dargestellt:



Schreiben Sie eine Prozedur `listToCons: int list -> cons`, die zu einer Liste die zugehörige Repräsentation mit Cons-Zellen erzeugt.

(d) Schreiben Sie eine Prozedur `isList: cons -> bool`, die testet, ob die durch eine Cons-Zelle repräsentierte Datenstruktur eine Liste ist. Die Prozedur darf divergieren, falls das nicht der Fall ist.

(e) Man kann auch sogenannte Kreise repräsentieren:



Schreiben Sie eine Prozedur `circle: cons -> unit`, die die durch eine Cons-Zelle repräsentierte Liste zu einem Kreis verkettet. Die Prozedur darf divergieren, wenn die Datenstruktur keine Liste ist.

(f) Schreiben Sie eine Prozedur `isCircle: cons -> bool`, die testet, ob die durch eine Cons-Zelle repräsentierte Datenstruktur ein Kreis ist. Die Prozedur darf divergieren, falls das nicht der Falls ist.

Bemerkung: Das Konzept der Cons-Zelle wurde in den 50er-Jahren für die Sprache Lisp eingeführt. In Lisp heißen die Prozeduren `fst` und `snd` aus historischen Gründen `car` (*content of address register*) und `cdr` (*content of data register*).