Technische Universität München
Fakultät für Informatik
Prof. Dr. A. Knoll

WS 2005/2006
Übungsblatt 0
23. Dezember 2005

Übungen zu Einführung in die Informatik I

Hinweis: Bei den folgenden Aufgaben handelt es sich um die Minimalanforderungen, die Prof. Knoll nach Ende der ersten Vorlesungshälfte stellt. Es handelt sich um Aufgaben, die in früheren Semestern teils als Klausuraufgabe gestellt wurden.

Aufgabe 1 Berechnung von π (Rekursive Funktionen)

Auf dem letzten Übungsblatt haben wir die Kreiszahl π definiert als 3.14. Im Lauf dieser Aufgabe werden wir versuchen, die Zahl π mit höherer Genauigkeit herzuleiten. Hierzu werden wir eine iteratives Verfahren benutzen, das auf dem folgendem Satz aufbaut:

Die Iteration $f_n = f_{n-1} + sin(f_{n-1})$ konvergiert (quadratisch) gegen den jeweils nächsten ganzzahligen Vielfachen von π

Zur Berechnung dieser Iteration leiten wir zuerst aus der bekannten Reihenentwicklung

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

eine rekursive Funktionsdefinition für $\sin x$ her.

a) Verwenden Sie diese Reihenformel, um eine Funktion `sin k x` zu definieren, die den Sinus von *x* durch die ersten *k* Glieder der unendlichen Reihe approximiert. Brechen Sie dazu die unendliche Summe nach *k* Summanden ab.

b) Testen Sie Ihre Approximation der Sinusfunktion mit einigen Werten für *x* sowie *k*. Finden Sie heraus, ab welchem *k* die Genauigkeit ausreichend ist.

c) Mit Hilfe dieser Funktion können Sie nun die Zahl π durch obige Iterationformel annähern. Verwenden Sie dazu den geeigneten Wert für *k* den Sie im vorherigen Schritt gefunden haben.

Aufgabe 2 Listen in OCaml

In dieser Aufgabe werden wir die Datenstruktur der Liste selbst definieren.

a) Definieren Sie rekursiv einen neuen Typ `myList` der entweder leer (`Empty`) sein kann oder ein Element (`Elem`) das aus einem Wert `'a` und einer Liste besteht.

b) Definieren Sie eine Funktion `prepend`, welche an eine vorhandene Liste einen Wert vorne anfügt. Testen Sie ihre neue Datenstruktur indem Sie eine Liste mit den Zeichen `a`, `h`, `a` aufbauen.

c) Schreiben Sie eine (rekursive) Funktion `append`, welche an eine vorhandene Liste einen Wert hinten anhängt. Testen Sie Ihre Funktion indem Sie das Zeichen `!` an die Liste der letzten Teilaufgabe anhängen.

d) Schreiben Sie eine (rekursive) Funktion `isElement`, welche für eine Liste und einen Wert testet, ob der Wert in der Liste enthalten ist. Testen Sie Ihre neudefinierte Funktion für die Liste aus den vorherigen Teilaufgaben und die Zeichen `x`, `h` und `a`.

e) Schreiben Sie eine (rekursive) Funktion `insert`, welche einen Wert in einer Liste **sortiert** einfügt. Die kleinsten Elemente sollen dabei am Ende der Liste stehen. Testen Sie Ihre Funktion mit den Werten 23, 12, 42, 24, 1, 23.

Aufgabe 3 Rekursive Datenstrukturen in OCaml

Warteschlangen (*PriorityQueue*) seien in dieser Aufgabe **sortierte** Listen, bei denen jedes Element neben einem Wert *(value)* beliebigen Typs noch eine natürliche Zahl als Priorität bzw. Schlüssel *(key)* enthält, nach dem **sortiert** wird.
Beispiel:

```
Elem(2, 'S', Elem(4,'P', Elem(5, 'Q', Elem(8, 'R', Empty))))
```

a) Definieren Sie in OCaml einen neuen Typ *PriorityQueue*.

b) Schreiben Sie eine OCaml-Funktion *extract*, welche einen Wert in einer Warteschlangen sucht, das Element welches den Wert enthält aus der Warteschlange entfernt und die Warteschlange ohne das Element zurückgibt.

c) Schreiben Sie eine OCaml-Funktion *merge*, welche zwei Warteschlangen vereinigt und die vereinigte Warteschlange als Ergebnis zurückgibt.

Aufgabe 4 Bäume

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.