Technische Universität München
Fakultät für Informatik
M. Bauer, D. Golias, S. Riesner,
Dr. M. Schneider

WS 2002/2003
Lösungsvorschläge zu Blatt 1
24. Oktober 2002

Übungen zu Einführung in die Informatik I

Aufgabe 1 Kaffeekochen funktional beschrieben

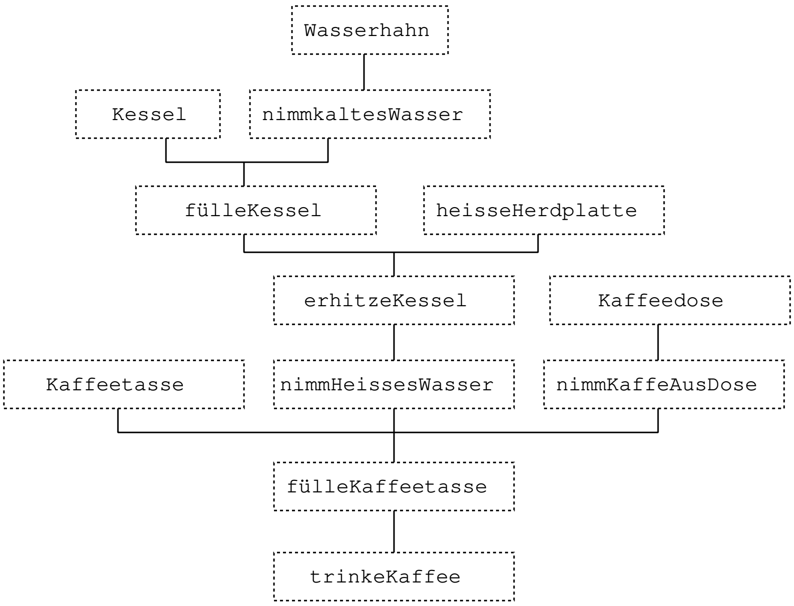
Die Aufgabe dient dazu den Studenten von Anfang an ein Gefühl zur funktionalen Algorithmenbildung / Programmierung zu vermitteln. Dazu soll die entsprechende Denkart an einem jedem nachvollziehbarem Beispiel textuell und informell eingeübt werden, dem Kaffeekochen.

Da bereits in der 1. Vorlesungsstunde ein entsprechendes Beispiel (Telefonat) durchgesprochen wird, ist den Studenten die Aufgabenart nicht neu.

Die Lösung ist rel. einfach. Dabei wollen wir hier UNBEDINGT davon ausgehen, dass alle Werkstoffe in ausreichender Menge verfügbar sind. Entsprechende Ausnahmen bleiben unberücksichtigt.

```
trinkeKaffee(fülleKaffetasse(Kaffeetasse, nimmKaffeAusDose(Kaffeedose),
nimmHeissesWasser(erhitzeKessel(fülleKessel(Kessel, nimmKaltesWasser),
heisseHerdplatte))))
```

bzw. graphisch struktoriert:



Aufgabe 2 Einfache Ausdrücke in OCaml

Die Lösung der Aufgabe besteht trivialerweise darin, die Ausdrücke des Aufgabenblattes im OCaml-Interpreter einzugeben. Sinn und Zweck ist es die (nativen) Operatoren einzuüben und ein Verständnis für die (interaktiven) Ausgaben des Interpreters zu entwickeln.

```
# 3 * 4 - (5 + 6) / 7 ;;
```

```
- : int = 11

# (-2 + (21 mod 9)) * 3 ;;
- : int = 3

# (true && not (false || true)) && not true ;;
- : bool = false

# (2 < 1) && (4 = 5) || (5 >= (29 mod 12)) ;;
- : bool = true
```

Besondere Beachtung verdienen dabei die ganzzahlige Division, die Modulo-Funktion, sowie die Möglichkeit Vergleichsoperationen auf komplexere Ausdrücke (vgl. 4. Beispiel) anzuwenden. Die Ergebnisse können zudem selbst wieder in boolschen Ausdrücken verwendet werden.

Die Ausdrücke beschränken sich auf die Datentypen bool und integer. Gleitkomma-Ausdrücke werden im Rahmen der Vorlesung und Zentralübung besprochen. Der wesentliche Unterschied besteht hierbei darin, dass für floats eigene native Operatoren existieren (+. -. /. *.). Die Operatoren müssen auf die Datentypen angewendet werden für die sie definiert wurden. Ein *Misch-Masch* (auch int und float) ist nicht gestattet. Explizites *Casten* ist möglich.

Aufgabe 3 Funktionen in OCaml

Für die Berechnung sich wiederholender Ausdrücke verwendet man in Programmiersprachen und damit auch in OCaml *Funktionen*.

Die Syntax wird am besten durch ein Beispiel ersichtlich. Starten Sie den OCaml Interpreter (`ocaml`) und geben sie den folgenden Befehl ein:

```
let f x = (x + 2) * (x - 2);;
```

An der Ausgabe erkennt man daß der Interpreter jetzt mit dem Wert `f` eine Funktion (`<fun>`) der natürlichen Zahlen (`int`) in die natürlichen Zahlen assoziiert hat. Die Funktion kann jetzt mit einer natürlichen Zahl als Parameter aufgerufen werden.

```
# let f x = (x - 2) * (x + 2);;
val f : int -> int = <fun>
# f 2;;
- : int = 0
```

Eine Funktion mit mehreren Parametern definiert man analog:

```
let f x y = (x + 1) * (y + 1);;
```

Auch diese Funktion kann jetzt mit zwei ganzen Zahlen als Parameter aufgerufen werden.

```
# let f x y = (x + 1) * (y + 1);;
val f : int -> int -> int = <fun>
# f 2 3;;
- : int = 12
```

Der Typ der Argumente wird automatisch aus der Funktionsdeklaration abgeleitet. Durch Verwendung der entsprechenden Operatoren erhält man eine Funktion der rationalen Zahlen in die rationalen Zahlen.

```
# let f x y = (x +. 1.0) *. (y +. 1.0);;
val f : float -> float -> float = <fun>
# f 1.3 4.2;;
- : float = 11.96
```

Verwendung eines Arguments vom falschen Typ führt zu einer Fehlermeldung:

```
# f 2 3;;
This expression has type int but is here used with type float
```

Genauso können auch Funktionen definiert werden, die keinen Parameter benötigen. Diese Funktionen geben einfach immer einen konstanten Wert zurück.

```
# let n = 42;;
val n : int = 42
# n;;
- : int = 42
```

a) **Definition von Funktionen:**

Definieren Sie eine einstellige Funktion `square` die zu einer Zahl x das Quadrat x^2 dieser Zahl zurückgibt. Testen Sie ihre neue Funktion mit einigen Werten. Definieren Sie weiter eine konstante Funktion `pi` die den Wert 3.14 zurückgibt.

```
# let square x = x *. x;;
val square : float -> float = <fun>
# square 2.0;;
- : float = 4.
# square 3.;;
- : float = 9.
# let pi = 3.14;;
val pi : float = 3.14
```

b) **Zusammengesetzte Funktionen:**

(i) Definieren Sie nun mit Hilfe der obigen Funktionen eine neue Funktion `circ_area` die zu einem Kreis mit Radius r den Flächeninhalt $A = \pi \cdot r^2$ berechnet. Testen Sie diese Funktion mit einigen Werten.

```
# let circ_area r = pi *. square r;;
val circ_area : float -> float = <fun>
# circ_area 1.0;;
- : float = 3.14
# circ_area 2.0;;
- : float = 12.56
```

(ii) Definieren Sie eine Funktion `cylinder_area` die die Oberfläche eines geraden Kreiszylinders mit Radius r und Höhe h berechnet als die Summe der beiden Deckflächen und der Mantelfläche ($2 \cdot \pi \cdot r \cdot h$). Testen Sie ihre Funktion mit einigen Werten.

```
# let cylinder_area r h = 2. *. (circ_area r +. pi *. r *. h);;
val cylinder_area : float -> float -> float = <fun>
# cylinder_area 1. 1. ;;
- : float = 12.56
```

(iii) Berechnen Sie mit Hilfe einer Funktion das Volumen eines geraden Kreiskegelstumpfes mit Höhe h und den beiden Radien r_1 und r_2 nach der Formel

$$V = \frac{h \cdot \pi}{3}({r_1}^2 + r_1 \cdot r_2 + {r_2}^2)$$

für verschiedene Werte.

```
#  let v r1 r2 h =
   (h *. pi *. (square r1 +. r1 *. r2 +. square r2))/. 3.;;
val v : float -> float -> float -> float = <fun>
# v 1. 2. 3.;;
- : float = 21.98
```

c) **Partielle Funktionsanwendung:** Nimmt man eine$n$-stellige Funktion und wendet sie auf ein einzelnes Argument an, so erhält man eine $(n-1)$-stellige Funktion. Die Funktion 'schluckt' das erste Argument. Dies wird an Ausgabe eindeutig ersichtlich: während `add` eine zweistellige Funktion ist, wird `inc` zu einer einstelligen Funktion.

```
# let sum a b = a + b;;
val sum : int -> int -> int = <fun>
# sum 3 ;;
- : int -> int = <fun>
# let inc = sum 1;;
val inc : int -> int = <fun>
# inc 0;;
- : int = 1
# inc ( inc ( 0 ) ) ;;
- : int = 2
# inc ( inc ( inc ( 0 ) ) ) ;;
- : int = 3
```

(i) Definieren Sie basierend auf der Funktion `cylinder_area` von oben eine neue Funktion, die die Oberfläche eines Zylinders mit Radius 1 berechnet.

(ii) Vergleichen Sie das Ergebnis mit der ursprünglichen Funktion. Vergewissern Sie sich, daß die neue Funktion tatsächlich ein Argument 'geschluckt' hat.

```
# let unit_cylinder_area = cylinder_area 1. ;;
val unit_cylinder_area : float -> float = <fun>
# unit_cylinder_area 2. ;;
- : float = 18.84
# cylinder_area 1. 2. ;;
- : float = 18.84
```