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

WS 2002/2003
Lösungsvorschläge zu Blatt 2
31. Oktober 2002

Übungen zu Einführung in die Informatik I

Aufgabe 5 Geschachtelte Funktionen und Rekursion (Lösungsvorschlag)

a) Unter Verwendung von Pseudocode lautet eine derartige Funktion

fct add = (**nat** x, **nat** y) **nat**:
  $x + y \cdot y$

In OCAML schreiben wir:

```
let add (x, y) = x + y*y;;
```

b) Die gesuchten Konstanten lassen sich durch OCAML-Ausdrücke wie folgt berechnen:

```
let c_0 = 0;;
let c_1 = add(0, 1);;
let c_2 = add(add(0, 1), 2);;
let c_3 = add(add(add(0, 1), 2), 3);;
```

c) Alternativ zur vorhergehenden Teilaufgabe läßt sich `c_2` und `c_3` auch folgendermaßen definieren:

```
let c_1 = add(c_0, 1);;
let c_2 = add(c_1, 2);;
let c_3 = add(c_2, 3);;
```

d) Wie in der Teilaufgabe c) beschrieben, läßt sich die Berechnung der Konstanten `c_n` auf die Berechnung von `c_n-1` zurückführen. Speziell muß der Fall $n = 0$ behandelt werden ($g(n) = 0$). Dies führt zu folgendem Algorithmus, den wir unter Verwendung von Pseudocode angeben:

fct g = (**nat** n) **nat**:
  if (n $\stackrel{?}{=}$0) **then** 0
      else add(g(n-1), n)

e) In OCAML lautet die Funktion:

```
let add (x, y) = x + y*y;;

let rec g n =
      if (n==0) then 0
               else add((g (n-1)), n);;
```

f) Der Prozess der Berechnung von $g(3)$ wird durch folgende Ausdrücke beschrieben:

```
g(3)
add(g(2) , 3)
add(add(g(1), 2), 3)
add(add(add(g(0), 1), 2), 3)
add(add(add(0, 1), 2), 3)
add(add(1, 2), 3)
add(5, 3)
14
```

Aufgabe 6 Wächter in OCaml (Lösungsvorschlag)

a) Die Aufgabe dient zwei Zwecken:

- Einübung der Syntax
- Darstellung der deutlichen Vorteile des Pattern-Matchings gegenüber der *i*f-then-else Strategie der imperativen Sprachen

Dabei gilt: Die nicht definierten Fälle lassen sich mit der `OCaml`-Möglichkeit Fehlermeldungs-Ausnahmen zu werfen, behandeln.

Version 1: Einfaches Dreieck mit *if-then-else*:

```
let f x = if (x < 0.) then failwith "undefined"
else if (x <= 0.75) then 2. *. x else if (x <1.5)
then 2. *. (1.5 -. x) else failwith "undefined"
;;
```

Version 2: Einfaches Dreieck mit *Pattern-Matching*

```
let f x = match x with
| n when ((n >= 0.) && (n < 0.75)) -> 2. *. x
| n when ((n >= 0.75) && (n < 1.5)) -> 2. *. (1.5 -. x)
| _ -> failwith "undefined"
;;
```

Das _ Muster dient hierbei als *Wildcard* für alle Muster die von den anderen Pattern nicht erkannt werden.

b) Die Periode kann durch die Verwendung der Modulo-Funktion erreicht werden. Dabei ist es jedoch notwendig eine Version für floats zu entwerfen:

```
let fmod x m = x -. floor( x /. m) *. m;;
```

Zu beachten: Statt des doppelten casts (vgl. Tutorbesprechung) wird in der Definition von fmod nun floor verwendet !

Version 3: Periodisches Dreieck, mit *if-then-else*

```
let f x = if ( (fmod x 1.5) < 0.) then failwith "undefined"
else if ( (fmod x 1.5) <= 0.75) then 2. *. (fmod x 1.5)
else if ( (fmod x 1.5) <1.5) then 2. *. (1.5 -. (fmod x 1.5))
else failwith "undefined"
;;
```

Die *undefined* Fälle könnten auch weggelassen werden, da sie bei ordnungsgemäßer Funktionsweise der fmod-Funktion nicht erreicht werden.

Version 4: Periodisches Dreieck, mit *Pattern-Matching*

```
let f x = match x with
| n when (( (fmod n 1.5) >= 0.)  && ( (fmod n 1.5) < 0.75))
-> 2. *. (fmod n 1.5)

| n when (( (fmod n 1.5) >= 0.75) && ( (fmod n 1.5) < 1.5))
-> 2. *. (1.5 -. (fmod n 1.5))

| _ -> failwith "undefined"
;;
```

Auch hier ist das Wildcard eigentlich unnötig, jedoch findet der Interpreter das Pattern-Matching als *möglicherweise* unzureichend:

Warning: Bad style, all clauses in this pattern-matching are guarded.

Allgemein zu beachten: Negative Funktionswerte sind zu klammern (OCaml-Typinferenz-Frage), also

```
f (-4.5);;

statt

f -4.5;;
```

verwenden.

c) Neben der weiteren Einübung des `OCaml`-PatternMatching werden in dieser Aufgabe erstmals Tupel im Rahmen der Übungen eingeführt.

Die naheliegendste Lösung geht von einer 1:1 Abbildung der gegebenen Wahrheitstafel aus.

```
let xor v = match v with
| (true, true) -> false
| (false, true) -> true
| (true, false) -> true
| (false, false) -> false
;;
```

Bei genauerer Betrachtung fällt auf, dass bei einer (Muster-)Übereinstimmung des 1. Tupel-Elementes das 2. das Ergebnis bestimmt. Eine kürzere Lösung ergibt sich demnach aus:

```
let xor v = match v with
| (false,x) -> x
| (true, x) -> not x
;;
```

d) Nach den Tupel werden nun Listen im Rahmen der Übungen eingeführt. Wesentlich ist hier, die *M*uster-Erkennung der unterschiedlichen Anzahl der Listenelemente.

   Beachten Sie: An dieser Stelle ist in der Vorlesung die Rekursion noch nicht eingeführt und kann folglich nicht verwendet werden.

```
let max a b = if (a < b) then b else a

let maxof4 v = match v with
| [a;b;c;d] -> max a (max b (max c d))
| [a;b;c] -> max a (max b c)
| [a;b] -> max a b
| [a] -> a
| [] -> failwith "No elements"
| _ -> failwith "To many elements"
;;
```

Aufgabe 7 Berechnung von π (Rekursive Funktionen)

Auf dem letzten Übungsblatt haben wir die Kreiszahl π definiert als 3.14. Dieser Wert ist natürlich nicht ganz richtig.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.

   In der Vorlesung und der Zentralübung wurde die Funktion `fac n` *erwähnt, die zu einer Zahl n die Fakultät*

$$n! = \prod_{k=1}^{n} k = n \cdot (n-1)!$$

   berechnet.

```
let rec fac n =
  if n=0.
  then 1.
  else n *. fac (n-.1.);;
```

Beachten Sie hier, daß die Fakultätsfunktion, die üblicherweise als vom Typ `int->int` *definiert wird hier als* `float->float` *verwendet wird. Das liegt daran, daß bei der Verwendung von 16bit Integer bereits bei* `fac 13` *ein Überlauf stattfinden würde; dies tritt in der Sinusreihe bereits im 5. Glied auf. Auch die hier verwendete Implementierung ist numerisch alles andere als einwandfrei, da auch bei Gleitkommazahlen die Genauigkeit beschränkt ist. Abhilfe würde die Verwendung von 32bit oder 64bit Integer Zahlen (oder auch unbeschränkte Ganzzahlarithmetik) schaffen, was allerdings die Verwendung von Modulen beinhalten würde und daher den Rahmen dieses zweiten Übungsblattes sprengen würde.*

Beachten Sie weitehin die Abbruchbedingung `n=0`*; alternativ könnte man auch* `n<0` *schreiben um auch für Werte kleiner 0 zu terminieren; allerdings würde dann der oben erwähnte Überlauf zu falschen Ergebnissen führen anstatt wie in der vorliegenden Version zu einem offensichtlichen Fehler.*

```
let rec sin k x =
  if k=0.
  then x
  else (-1.)**k *. x**(2.*.k +. 1.) /. fac(2.*.k +. 1.)
  +. sin (k-.1.) x ;;
```

Alternativ auch mit k als Integer, aber unschön durch viele Cast-Operationen:

```
let rec sin k x =
  if k=0
  then x
  else (-1.)**float_of_int(k)
    *. x**float_of_int(2*k + 1) /. fac( float_of_int(2*k + 1))
    +. sin (k-1) x ;;
```

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

Informell: Bei einer streng monoton konvergierenden Reihe (ohne Beweis) kann man die Genauigkeit ablesen an der Zahl der Ziffern die sich ändern.

```
# sin 5 1.0;;
- : float = 0.841470984648
# sin 6 1.0;;
- : float = 0.841470984809
# sin 7 1.0;;
- : float = 0.841470984808
# sin 8 1.0;;
- : float = 0.841470984808
```

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.

```
let rec pi k =
  if k = 0.
  then 1.
  else sin 10. ( pi (k -. 1.) ) +. pi (k -. 1.);;
```

Alternativ auch mit k als Integer:

```
let rec pi k =
  if k = 0.
  then 1.
  else sin 10 (pi (k-1)) +. pi (k-1);;
```

Aufgabe 8 (H) Berechnung von π nach dem Monte-Carlo Verfahren

Berechnungsmethoden die Zufallszahlen und Wahrscheinlichkeiten verwenden, werden oft *Monte-Carlo-Verfahren* genannt. Im Folgenden berechnen wir π mit einem Monte-Carlo-Verfahren.

Die Fläche eines Kreises ist bekanntlich durch die Formel $A = \pi \cdot r^2$ gegeben; für einen Kreis mit Radius 1 ergibt sich somit der Flächeninhalt π. Wenn wir nun zufällig einen Punkt (x, y), $0 \leq x \leq 1 \wedge 0 \leq y \leq 1$ wählen, dann ist die Wahrscheinlichkeit, dass dieser Punkt innerhalb des Einheitskreises liegt, gegeben durch:

$$P(x^2 + y^2 < 1) = \frac{A_{kreis}}{A_{quadrat}} = \frac{\pi}{4}.$$

Nun wähle man N Punkte (x, y), $0 \leq x \leq 1 \wedge 0 \leq y \leq 1$; M bezeichne die Anzahl der Punkte, die innerhalb des Einheitskreises liegen; dann gilt für sehr große N die Näherung:

$$\pi = \frac{4 \cdot M}{N}$$

a) Berechnen Sie π nach dem beschriebenen Verfahren mit Hilfe von OCaml. (**Hinweis:** Definieren Sie zunächst eine boolesche Hilfsfunktion `inside (x, y)`, die ermittelt, ob der Punkt (x, y) innerhalb des Kreise liegt.)

```
let inside x y = if (x*.x +. y*.y) < 1.
  then true
  else false;;
```

Beachten Sie, daß Sie die Wurzel, die eigentlich in der Formel des Pythagoras steht, beim Einheitskreis nicht benötigen.

b) Definieren Sie eine Zählfunktion `innerhalb (n)` zur Berechnung von M. Berechnen Sie nun π. (**Hinweis:** Eine gleichverteilte Zufallszahl aus dem Interval $[0, 1]$ erhalten sie in OCaml mit der Funktion `Random.float 1.0`.)

```
let rec innerhalb n =
  if n=0
  then 0
  else
    if inside (Random.float 1.) (Random.float 1.)
    then innerhalb (n-1) + 1
    else innerhalb (n-1);;
```

Alternative Lösung, eleganter, aber nicht ganz so sauber wegen Verwendung von 0 und 1 als bool-Werte:

```
let inside x y = if (x*.x +. y*.y) < 1.
  then 0
  else 1;;

let rec innerhalb n =
  if n=0
  then 0
  else innerhalb (n-1) + inside (Random.float 1.) (Random.float 1.);;
```

Berechnen Sie nun $\pi = \frac{4 \cdot M}{N}$ und testen Sie die Funktion mehrfach mit denselben Eingabewerten.

```
let pi n = 4. *. (float_of_int(innerhalb n)/. float_of_int(n));;

# pi 10000;;
- : float = 3.142
# pi 10000;;
- : float = 3.1652
# pi 10000;;
- : float = 3.1472
```

Eine alternative schöne Lösung sieht so aus:

```
# let inside v = match v with
        |(x,y) when ((x**2.+.  y**2.)<=1.) -> true
        |_ -> false;;
  val inside : float * float -> bool = <fun>

# let rec innerhalb k v = match v with
        |(m,n) when   n=k -> (m,n)
        |(m,n) when (inside (Random.float 1.0, Random.float 1.0))
                -> innerhalb k (m+1,n+1)
        |(m  ,n) -> innerhalb k (m,n+1);;
val innerhalb : int -> int * int -> int * int = <fun>

# let pi_e (m,n) = 4.*.(float_of_int m /. float_of_int n);;
val pi_e : int * int -> float = <fun>

# let pi k = pi_e (innerhalb k (0,0));;
val pi : int -> float = <fun>

# pi 10000;;
- : float = 3.1452

#   pi 10000;;
- : float = 3.1336
```

Mehr Informationen über π und weitere Näherungsverfahren finden Sie auch im Internet unter http://mathworld.wolfram.com/Pi.html