Technische Universität München
Fakultät für Informatik
C. Buckl, D. Gmach, K. Schlüter,
Dr. M. Schneider

WS 2005/2006
Lösungsvorschläge zu Blatt 3
11. November 2005

Übungen zu Einführung in die Informatik I

Aufgabe 9 Funktionensdefinitionen in OCaml (Lösungsvorschlag)

a) Eine namenlose Funktion wird in OCaml mit dem Schlüsselwort `fun` definiert. Natürlich kann man dieser Funktion auf einen Namen (`incr`) zuweisen.

```
# fun x -> x + 1;;
- : int -> int = <fun>
# (fun x -> x + 1) 4;;
- : int = 5
# let incr = fun x -> x + 1;;
val incr : int -> int = <fun>
# incr 4;;
- : int = 5
```

b) Streng genommen gibt es in OCaml nur einstellige, namenlose Funktionen der Form `fun` *param* `->` *expr*. Alles andere sind abkürzende Schreibweisen. Die Signatur der `sum` Funktion sieht streng geklammert folgendermaßen aus: `val sum : int -> (int -> int) = <fun>`.

```
# let sum = (fun x -> (fun y -> x + y));;
val sum : int -> int -> int = <fun>
# sum 3 5;;
- : int = 8
# sum 3;;
- : int -> int = <fun>
# (sum 4) 5;;
- : int = 9
```

Funktionen mit mehreren Argumenten stellen eine abkürzende Schreibweise dar und werden durch geschachtelte Funktionen ausgedrückt. Dies wird auch als *Currying* bezeichnet.

```
# let sum = fun x y -> x + y;;
val sum : int -> int -> int = <fun>
```

Eine weitere abkürzende Schreibweise bei benannten Funktion ist die Möglichkeit das Schlüsselwort `fun` wegzulassen und stattdessen die Parameter direkt hinter dem Namen der Funktion zu schreiben.

```
# let sum x y = x + y;;
val sum : int -> int -> int = <fun>
```

c) Das Schlüsselwort `function` verhält sich bei einstelligen Funktionen wie `fun`. Hauptsächlich ergeben sich zwei Unterschiede:

- Mit `function` können nur einstellige Funktionen definiert werden.

```
# let sum = function x -> (function y -> x + y);;
val sum : int -> int -> int = <fun>
# let sum = function x y -> x + y;;
Characters 21-22:
  let sum = function x y -> x + y;;
                        ^
Syntax error
```

- Mit `function` kann zusätzlich nach dem Parameter ein Pattern Matching durchgeführt werden und stellt somit eine abkürzende Schreibweise für Funktionen mit Patternmatching dar.

```
# let rec fib = fun i -> match i with
          0 -> 0
        | 1 -> 1
        | j -> fib (j - 2) + fib (j - 1);;
val fib : int -> int = <fun>
# let rec fib2 = function
          0 -> 0
        | 1 -> 1
        | j -> fib2 (j - 2) + fib2 (j - 1);;
val fib2 : int -> int = <fun>
```

Aufgabe 10 Arbeiten mit Modulen (Lösungsvorschlag)

Zur besseren Übersicht und um gleichnamige Operationen für verschiedenen Datentypen zu unterscheiden verwendet OCaml sogenannte *Module*. Ein Beispiel ist: `Random.float`, das die Funktion *float* aus dem Modul *Random* aufruft.

In dieser Aufgabe verwenden wir Funktionen aus dem Modul *List* zur Operation auf Listen von Datentypen.

a) Definieren und belegen Sie in OCaml die beiden Zeichensequenzen `vorname` und `nachname` jeweils mit Ihrem Vor- bzw. Nachnamen. Verknüpfen Sie anschließend diese beiden Zeichensequenzen zu einer einzigen Zeichensequenz `name` unter Verwendung des Konkatenationsoperator `@` und einem Komma sowie einem Leerzeichen zwischen den beiden Namensteilen.

```
# let nachname = ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd']
  and vorname = ['w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm'];;
val nachname : char list = ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd']
val vorname : char list = ['w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm']


# let name = nachname @ [','] @ [' '] @ vorname;;
val name : char list =
  ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd'; ','; ' ';
   'w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm']
```

b) Verwenden Sie die Funktion *length* aus dem Modul *List* um die Länge des ganzen Namens festzustellen.

```
# List.length name;;
- : int = 16
```

c) Revertieren Sie Ihren (oben definierten) Namen mit Hilfe von OCaml durch Zerlegung und unter ausschließlicher Verwendung der Funktionen *hd* und *tl* aus dem Modul *List* und des Operators `::` (‘Prepend’).
Hinweis: Zur Vereinfachung der Schreibweise bietet es sich an zuerst zwei Funktionen `head` und `tail` zu definieren, welche genau die entsprechenden Funktionen aus dem Modul *List* aufrufen.

```
# let head = List.hd;;
val head : 'a list -> 'a = <fun>
# let tail = List.tl;;
val tail : 'a list -> 'a list = <fun>

# head(tail(tail(tail(tail(tail(tail(name)))))));
head(tail(tail(tail(tail(tail(name))))));
head(tail(tail(tail(tail(name)))));
head(tail(tail(tail(name))));
head(tail(tail(name)));
head(tail(name));
[head(name)]
;;
- : char list = ['d'; 'n'; 'a'; 'r'; 'g'; 'e'; 'l']
```

d) Definieren Sie die folgende *Cast*-Funktion `list_of_string` die mit Hilfe von Funktionen aus den Modulen *String* und *Stream* aus einem String eine Liste von Einzelzeichen macht.

```
let list_of_string s =
  Stream.npeek (String.length s) (Stream.of_string s);;
```

Testen Sie diese Funktion mit ein paar Eingabestrings.

```
# list_of_string("evitez les courrants d'air");;
- : char list =
['e'; 'v'; 'i'; 't'; 'e'; 'z'; ' '; 'l'; 'e'; 's'; ' ';
 'c'; 'o'; 'u'; 'r'; 'r'; 'a'; 'n'; 't'; 's'; ' '; 'd'; '\''; 'a'; 'i'; 'r']
```

e) Definieren Sie nun eine rekursive Funktion *revert*, die eine beliebige Liste umkehrt.
Testen Sie diese Funktion für Ihren oben definierten Namen sowie für den String `reliefpfeiler` (unter Verwendung der Cast-Funktion aus Teilaufgabe c).

```
let rec revert l = match l with
  | []    -> []
  | a::l  -> revert l @ [a];;
```

```
# revert name;;
- : char list =
['m'; 'a'; 'i'; 'l'; 'l'; 'i'; 'w'; ' '; ',';
 'd'; 'n'; 'a'; 'r'; 'g'; 'e'; 'l']

# revert (list_of_string("reliefpfeiler"));;
- : char list =
['r'; 'e'; 'l'; 'i'; 'e'; 'f'; 'p'; 'f'; 'e'; 'i'; 'l'; 'e'; 'r']
```

Aufgabe 11 Komplexe Zahlen (Lösungsvorschlag)

a)
```
# type complex = Complex of float * float;;
type complex = Complex of float * float
# Complex(3.0,5.5);;
- : complex = Complex (3., 5.5)
```

b)
```
# let add (Complex(x,y)) (Complex(u,v)) = Complex(x +. u, y +. v);;
val add : complex -> complex -> complex = <fun>
# add (Complex(3.0,5.5)) (Complex(1.0,1.0));;
- : complex = Complex (4., 6.5)
```

c)
```
# let (++) (Complex(x,y)) (Complex(u,v)) = Complex(x +. u, y +. v);;
val ( ++ ) : complex -> complex -> complex = <fun>
# (Complex(3.0,5.5)) ++ (Complex(1.0,1.0));;
- : complex = Complex (4., 6.5)
```

Hinweis: In OCaml gibt es kein Überladen von Funktionen und Operatoren (d.h.: es gibt nicht mehrere Definitionen einer Funktion oder eines Operators für verschiedene Datentypen). Der Operator ++ ist in OCaml standardmäßig nicht vorhanden. Gäbe es ihn würde er durch diese Aufgabe überschrieben werden.

d)
```
# type complex = Complex of float * float
               | Polar of float * float;;
# let pi = 2. *. asin 1.;;
# let polar c = match c with
        Complex(x,y) ->
            let r = sqrt(x *. x +. y *. y) in
            let psi = if y < 0. then acos((-. x) /. r) -. pi
                                else acos(x /. r) in
            Polar(r,psi)
      |  _ -> c;;
val polar : complex -> complex = <fun>
# polar (Complex(3.0,5.5));;
- : complex = Polar (6.2649820430708338, 1.0714496051147666)
# polar (Complex(0.0,1.0));;
- : complex = Polar (1., 1.5707963267948966)
# polar (Polar(1., 1.));;
- : complex = Polar (1., 1.)
# let normal c = match c with
         Polar(r,psi) ->
```

```
                    let x = r *. cos psi in
                    let y = r *. sin psi in
                    Complex(x,y)
            | _ -> c;;
    val normal : complex -> complex = <fun>
    # normal (Polar(1., 0.));;
    - : complex = Complex (1., 0.)
    # normal(polar(Complex(1.0,1.0)));;
    - : complex = Complex (1., 1.0000000000000002)
```

e)
```
# let rec (++) a b = match (a,b) with
        (Complex(x,y),Complex(u,v)) -> Complex(x +. u, y +. v)
        | (_,Polar(_,_)) -> a ++ (normal(b))
        | (Polar(_,_),_) -> (normal(a)) ++ b;;
val ( ++ ) : complex -> complex -> complex = <fun>
```

Alternativ:

```
# let rec add a b = match (a,b) with
        (Complex(x,y),Complex(u,v)) -> Complex(x +. u, y +. v)
        | (_,Polar(_,_)) -> add a (normal(b))
        | (Polar(_,_),_) -> add (normal(a)) b;;
val add : complex -> complex -> complex = <fun>
# let (++) = add;;
val ( ++ ) : complex -> complex -> complex = <fun>
# (Complex(3.0,5.5)) ++ (polar(Complex(1.0,1.0)));;
- : complex = Complex (4., 6.5)
```

Aufgabe 12 Kalender des Jahres 2006 (Lösungsvorschlag)

Eine mögliche Lösung könnte folgendermaßen aussehen:

```
type monat = Januar | Februar | März | April | Mai
           | Juni | Juli | August | September
           | Oktober | November | Dezember;;

type datum = Datum of (int * monat * int);;

type wochentag = Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag;;

let schaltjahr (jahreszahl) = (jahreszahl mod 4 = 0) &&
                              not (jahreszahl mod 100 = 0) ||
                             (jahreszahl mod 400 = 0);;

let monatslaenge (monat, jahr)  = match monat with
        |Januar | März | Mai | Juli | August |
         Oktober | Dezember -> 31
        |Februar when (schaltjahr jahr) -> 29
        |Februar -> 28
        |_ -> 30;;

let naechsterWochentag tag =
    let tagesliste = [Montag;Dienstag;Mittwoch;Donnerstag;Freitag;Samstag;Sonntag] in
```

```
      let rec naechsterWochentagEm (day, list) = match (day, list) with
        |(d, first :: rest) when (first <= d) -> naechsterWochentagEm (d, rest)
        |(d, first :: rest) -> first
        |(d, []) -> Montag
    in naechsterWochentagEm (tag, tagesliste);;

let naechsterMonat monat =
    let monatsliste = [Januar;Februar;März;April;Mai;Juni;Juli;August;September;
                       Oktober;November;Dezember] in
      let rec naechsterMonatEm (month, list) = match (month, list) with
        |(m, first :: rest) when (first <= m) -> naechsterMonatEm (m, rest)
        |(m, first :: rest) -> first
        |(m, []) -> Januar
      in naechsterMonatEm (monat, monatsliste);;

let tageSeitNeujahr2006 datum =
    let rec tageSeitNeujahrEm (datum, monat) = match datum with
       |Datum (t, m, j) when m = monat -> 0
       |Datum (t, m, j) ->
          monatslaenge (monat, j) + tageSeitNeujahrEm (datum, naechsterMonat (monat)) in
     match datum with
       |Datum (t, m, j) -> t-1 + tageSeitNeujahrEm (datum, Januar);;

let wochentagZu datum = let offset = (tageSeitNeujahr2006 datum) mod 7
        and neujahrstag = Sonntag in
        let rec zaehleTage (neujahrstag, n) = match n with
          |0 -> neujahrstag
          |n -> zaehleTage ((naechsterWochentag neujahrstag), n-1)
        in zaehleTage (neujahrstag, offset);;

let kalender2006 (monat) =
   let jahr = 2006 in
     let rec kalenderEm (monat, tage) = match tage with
       |tage when (tage = 0) -> []
       |tage -> ((wochentagZu (Datum(tage, monat, jahr))), Datum(tage,monat, jahr)) ::
                 kalenderEm (monat, tage-1)in
       kalenderEm (monat, monatslaenge(monat, jahr));;
```