TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen
Prof. Dr. Helmut Seidl, T. M. Gawlitza, S. Pott, M. Schwarz

WS 2008/09
Übungsblatt 6
18.11.2008

Abgabe: 25.11.2008 (vor der Vorlesung)

Aufgabe 6.1 (H) Rekursive Funktion

Gegeben sei folgendes MiniJava-Programm:

```
int x, y;

void log() {
  if (x!=1) {
    if (x%2==0) {
      x=x/2;
      log();
      y=y+1;
    }
    else {
      x=x-1;
      log();
    }
  } else {
    y=0;
  }
}
```

a) Erstellen Sie das Kontrollfluss-Diagramm für die Prozedur `log()`!

b) Zeigen Sie die Gültigkeit des Tripels

$$\{1 \leq 2^m \leq \mathtt{x} < 2^{m+1}\} \quad \mathtt{log}(); \quad \{\mathtt{y} = m\},$$

wobei m eine logische Variable ist.

c) Übersetzen Sie die Funktion `log()` möglichst genau in OCaml.

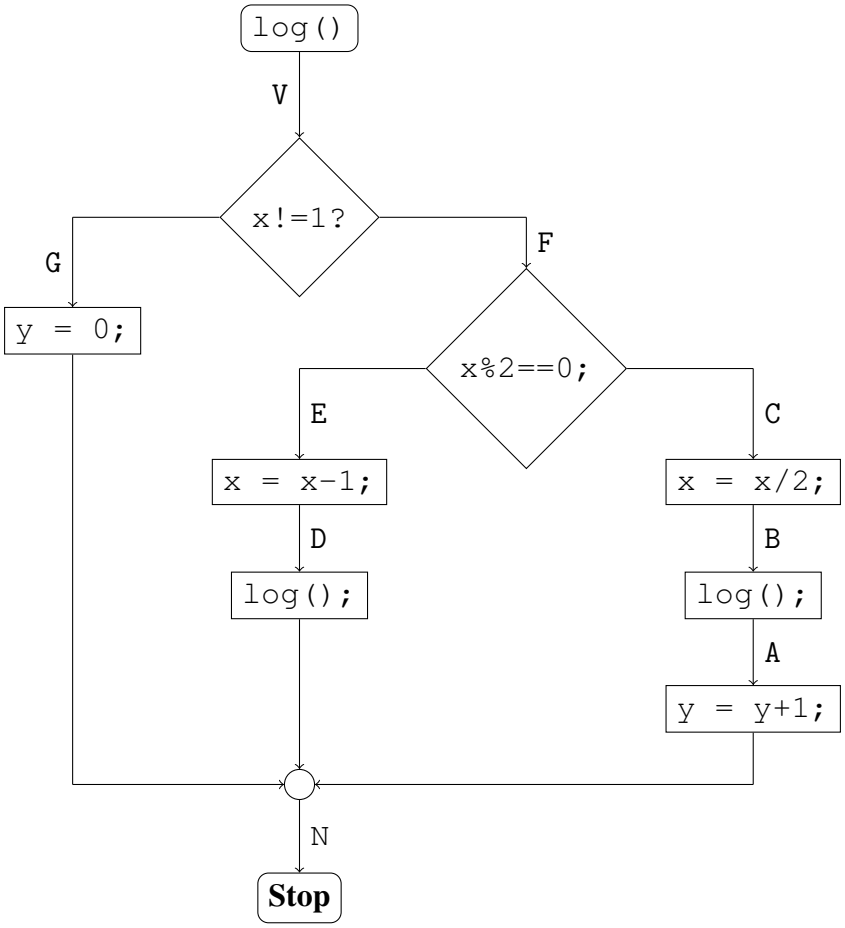
Lösungsvorschlag 6.1

a) s. b)

b) Wir setzen:

$$\mathtt{V} :\equiv 1 \leq 2^m \leq \mathtt{x} < 2^{m+1}$$
$$\mathtt{N} :\equiv \mathtt{y} = m$$

Wir erstellen zunächst das Kontrollfluß-Diagramm:



Los geht's:

$$\mathbf{WP}[\![\mathtt{y} = 0;]\!](\mathtt{N}) \equiv m = 0 \equiv: \mathtt{G}$$

$$\mathbf{WP}[\![\mathtt{y} = \mathtt{y} + 1;]\!](\mathtt{N}) \equiv \mathtt{y} = m - 1 \equiv: \mathtt{A}$$

Wir setzen:

$$\mathtt{B} :\equiv 1 \leq 2^{m-1} \leq \mathtt{x} < 2^m$$

$$\mathtt{D} :\equiv 1 \leq 2^m \leq \mathtt{x} < 2^{m+1}$$

Weiter geht's:

$$\begin{aligned}\mathbf{WP}[\![\mathtt{x} = \mathtt{x}/2;]\!](\mathtt{B}) &\equiv 1 \leq 2^{m-1} \leq \mathtt{x}\ \mathbf{div}\ 2 < 2^m \\ &\Leftarrow 2 \leq 2^m \leq \mathtt{x} < 2^{m+1} \equiv: \mathtt{C}\end{aligned}$$

$$\mathbf{WP}[\![\mathtt{x} = \mathtt{x} - 1;]\!](\mathtt{D}) \equiv 1 \leq 2^m \leq \mathtt{x} - 1 < 2^{m+1} \equiv: \mathtt{E}$$

Die Verzweigungen:

$$\begin{aligned}\mathbf{WP}[\![\mathtt{x}\%2 == 0;]\!](\mathtt{E},\mathtt{C}) &\equiv ((\mathtt{x}\ \mathbf{mod}\ 2 = 1) \wedge (1 \leq 2^m \leq \mathtt{x} - 1 < 2^{m+1})) \\ &\quad \vee ((\mathtt{x}\ \mathbf{mod}\ 2 = 0) \wedge (2 \leq 2^m \leq \mathtt{x} < 2^{m+1})) \\ &\quad \Leftarrow (1 \leq 2^m \leq \mathtt{x} < 2^{m+1}) \equiv: \mathtt{F}\end{aligned}$$

$$\begin{aligned}\mathbf{WP}[\![\mathtt{x}!=1]\!](\mathtt{G},\mathtt{F}) &\equiv ((\mathtt{x} = 1) \wedge (m = 0)) \vee ((\mathtt{x} \neq 1) \wedge (1 \leq 2^m \leq \mathtt{x} < 2^{m+1})) \\ &\quad \Leftarrow (1 \leq 2^m \leq \mathtt{x} < 2^{m+1}) \equiv \mathtt{V}\end{aligned}$$

Dabei ist zu beachten:

i) Das Tripel $\{\texttt{B}\}\texttt{log}();\{\texttt{A}\}$ ist gültig, da $\texttt{B} \equiv \texttt{V}[m-1/m]$ und $\texttt{A} \equiv \texttt{N}[m-1/m]$ gelten. Es ist also durch Substitution der logischen Variablen m durch den Term $m-1$, der keine Programmvariablen enthält, entstanden.

ii) Das Tripel $\{\texttt{D}\}\texttt{log}();\{\texttt{N}\}$ ist gültig, da $\texttt{D} \equiv \texttt{V}$ gilt.

```
let rec log x =
  if x <> 1 then
    if (x mod 2 = 0) then
      1 + log (x/2)
    else
      log (x-1)
  else
    0
```

Aufgabe 6.2 (H) Finanzmärkte

In dieser Aufgabe wird Aufgabe 5.6 fortgeführt. Dabei wird davon ausgegangen, dass jedes Wertpapier höchstens einmal in einem Portfolio vorkommt. Beispielsweise ist das Portfolio

```
c) [(2,{bez="RWE";wert=66});(3,{bez="RWE";wert=66});
                (911,{bez="PORSCHE";wert=60})]
```

nicht erlaubt. Das Portfolio

```
[(5,{bez="RWE";wert=66});(911,{bez="PORSCHE";wert=60})]
```

ist hingegen erlaubt. Berücksichtigen Sie dies in den nachfolgenden Teilaufgaben.

a) Schreiben Sei eine Funktion `add_wertpapier`. Der Aufruf `add_wertpapier a w p` soll `a` viele `w`-Wertpapiere zum Portfolio `p` hinzufügen. Dabei sollen doppelte Einträge vermieden werden.

   Beispiel: Für

   ```
   p = [(2,{bez="RWE";wert=66});(911,{bez="PORSCHE";wert=60})]
   ```

   liefert der Aufruf

   ```
   add_wertpapier 5 {bez="RWE";wert=66} p
   ```

   den Wert

   ```
   p = [(7,{bez="RWE";wert=66});(911,{bez="PORSCHE";wert=60})]
   ```

   zurück.

b) Nutzen Sie die in Teil a) definierte Funktion `add_wertpapier`, um nachfolgend beschriebene Funktion `union_portfolios` zu definieren. Der Aufruf `union_portfolios p1 p2` soll ein Portfolio liefern, das aus allen Wertpapieren beider Portfolios `p1` und `p2` besteht.

c) Schreiben Sie eine Funktion `wertaenderung`. Der Aufruf `wertaenderung bez d p` soll ein Portfolio zurück liefern, das dem Portfolio `p` entspricht, bis darauf, dass der Wert des mit `bez` bezeichneten Wertpapiers um `d` Prozent erhöht ist.

d) Schreiben Sie eine Funktion `max`, die dasjenige Wertpapier innerhalb eines Portfolios bestimmt, in das am meisten Geld investiert ist. Schreiben Sie diese Funktion einmal in einer nicht end-rekursiven Version und einmal in einer end-rekursiven Version.

Lösungsvorschlag 6.2

```
(* Loesungen von Aufgabe 5.6 *)

type wertpapier = { bez : string; wert : int }

type kunde = { name : string; portfolio : (int * wertpapier) list }

let rec portfoliowert = List.fold_left (fun s (a,w) -> s + a * w.wert) 0

let kundenwert k = portfoliowert k.portfolio

let rec anzahl praedikat kunden_liste =
  match kunden_liste with
      [] -> 0
    | k::ks ->
        anzahl praedikat ks +
        if praedikat k then
          1
        else
          0

let filter = List.filter

let grosskunden = filter (fun k -> kundenwert k >= 1000000)

(* Ab hier sind die Loesungen zu den Hausaufgaben *)

let rec add_wertpapier a w p =
  match p with
      [] -> [(a,w)]
    | (a',w')::p ->
        if w = w' then
          (a + a', w)::p
        else
          (a',w')::add_wertpapier a w p

let rec union_portfilios p1 p2 =
  match p1 with
      [] -> p2
    | (a,w)::p1 -> union_portfilios p1 (add_wertpapier a w p2)

let rec aender bez d = function
    [] -> []
  | (a,w)::p ->
      if w.bez = bez then
        (a,{bez=w.bez;wert=w.wert+(w.wert * d)/100})::aender bez d p
      else
        (a,w)::aender bez d p

let rec max = function
    [] -> (0,{bez="";wert=0})
  | (a,w)::p ->
```

```
      let (a',w') = max p in
      if a * w.wert >= a' * w'.wert then
        (a,w)
      else
        (a',w')

let rec max (a',w') = function
    [] -> (a',w')
  | (a,w)::p ->
      if a * w.wert >=  a' * w'.wert then
        max (a,w) p
      else
        max (a',w') p

let max = max (0,{bez="";wert=0})
```

Aufgabe 6.3 (P) OCaml vs. Java 2!

Übersetzen Sie die folgende Java-Klasse möglichst genau in OCaml.

```
public class Java {
  static abstract class Tree {
    int value;
    abstract boolean isLeaf();
    abstract int sum_up();
    abstract int depth();
  }

  static class Leaf extends Tree {
    Leaf(int v) {
      value = v;
    }
    boolean isLeaf() {
      return true;
    }
    int sum_up() {
      return value;
    }
    int depth() {
      return 1;
    }
  }

  static class Node extends Tree {
    Tree l, r;
    Node(int v, Tree left, Tree right) {
      value = v;
      l = left;
      r = right;
    }
    boolean isLeaf() {
      return false;
    }
    int sum_up() {
      return value + l.sum_up() + r.sum_up();
    }
    int depth() {
      int dl = l.depth();
      int dr = r.depth();
      if (dl < dr)
        return 1 + dr;
      else
        return 1 + dl;
    }
  }
}
```

Lösungsvorschlag 6.3

```
type tree = Leaf of int | Node of tree * int * tree
```

```
let rec sum_up = function
    Leaf x -> x
  | Node(l,x,r) -> x + sum_up l + sum_up r

let rec depth = function
    Leaf _ -> 1
  | Node(l,_,r) ->
      let dl = depth l in
    let dr = depth r in
    if dl < dr then
      1 + dr
    else
    1 + dl
```

Aufgabe 6.4 (P) Mergesort

Mergesort ist ein rekursiver Sortieralgorithmus, der gemäß dem Divide-and-Conquer-Prinzip arbeitet. Hier soll Mergesort zum Sortieren von Listen implementiert werden. Die prinzipielle Arbeitsweise des Algorithmus läßt sich wie folgt skizzieren.

- Null- und einelementige Listen sind sortiert
- Listen mit mehr als einem Element werden in zwei möglichst gleich große Teillisten aufgeteilt. Die Teillisten werden durch einen rekursiven Aufruf von Mergesort sortiert und anschließend in einem Mischschritt zu einer sortierten Liste zusammengesetzt.

Um dieses Prinzip umzusetzen, sollen Sie folgende Funktionen definieren:

a) Eine Funktion `init : 'a list -> 'a list list`. Der Aufruf `init` $[x_1; x_2; \ldots; x_n]$ soll die Liste $[[x_1]; [x_2]; \ldots; [x_n]]$ liefern.

b) Die Funktion `merge : 'a list -> 'a list -> 'a list` soll den Mischschritt implementieren. Als Argumente erhält sie **bereits sortierte** Listen `l1` und `l2`. Als Ergebnis liefert sie die sortierte Liste aller Elemente aus `l1` und `l2` zurück.

Beispiel: `merge` $[1; 5; 9]\ [2; 3; 11] = [1; 2; 3; 5; 9; 11]$

c) Eine Funktion `merge_list : 'a list list -> 'a list list`. Der Aufruf

$$\texttt{merge\_list}\ [l_1; l_2; \ldots; l_n]$$

soll sich zu der Liste

$$[\texttt{merge}\ l_1\ l_2; \texttt{merge}\ l_3\ l_4; \ldots; \texttt{merge}\ l_{n-1}\ l_n]$$

falls n gerade ist und zu der Liste

$$[\texttt{merge}\ l_1\ l_2; \texttt{merge}\ l_3\ l_4; \ldots; \texttt{merge}\ l_{n-2}\ l_{n-1}; l_n]$$

andernfalls auswerten.

Beispiel:

$$\begin{aligned} &\texttt{merge\_list}\ (\texttt{merge\_list}\ [[1; 5; 9]; [2; 3; 11]; [4; 7; 8]]) \\ =&\texttt{merge\_list}\ [[1; 2; 3; 5; 9; 11]; [4; 7; 8]] \\ =&[[1; 2; 3; 4; 5; 7; 8; 9; 11]] \end{aligned}$$

d) Definieren Sie schließlich die Funktion `mergesort : 'a list -> 'a list` mit Hilfe der obigen Funktionen.

Lösungsvorschlag 6.4

```
let init l = List.map (fun x -> [x]) l

let rec merge l1 l2 =
  match (l1,l2) with
    ([],l) | (l,[]) -> l
  | (x::xs,y::ys) ->
      if x <= y then
        x::(merge xs l2)
```

```
      else
        y::(merge l1 ys)

let rec merge_list = function
    [] -> []
  | [l] -> [l]
  | l1::l2::ls -> (merge l1 l2)::(merge_list ls)

(* Alternative merge_list - Funktion *)

let rec merge_list acc = function
    [] -> acc
  | [l] -> l::acc
  | l1::l2::ls -> merge_list (merge l1 l2 :: acc) ls

let merge_list l = merge_list [] l

(* -- Alternative merge_list - Funktion *)

let rec mergesort l =
  match l with
    [] -> []
  | [x] -> x
  | _ -> mergesort (merge_list l)

let mergesort l = mergesort (init l)

(* Eine andere Moeglichkeit,
   die nicht in der Aufgabenstellung beschrieben ist. *)

let rec split (l1,l2) = function
  x::y::l -> split (x::l1,y::l2) l
| [x]     -> (x::l1,l2)
| _       -> (l1,l2)

let split l = split ([],[]) l

let rec sort l =
  match l with
    [] | [_] -> l
  | _ ->
      let (l1,l2) = split l in
      let (l1,l2) = (sort l1, sort l2) in
      merge l1 l2
```