

Musterlösung 11

Aufgabe 1 (Bäume als Datenspeicher)

a) Invariante

```

inv Text _ = True
inv Program _ = True
inv Directory [] = True
inv Directory ((n,f):l) = inv f && not( element n (map fst l) )
                        && inv l -- keine doppelten Dateinamen

```

Darüber hinaus wäre vorstellbar, in der Invariante zu fordern, dass die Einträge in einem Verzeichnis immer nach ihrem Namen sortiert sind, oder dass Dateinamen eine bestimmte Länge nicht überschreiten.

b) Spezifikation von `rename` durch Voraussetzung und Effektbeschreibung

```

rename :: File -> String -> String -> File
rename dir oldname newname
pre (Directory d) oldname newname = oldname `elem` (map fst d)
                                   && not (newname `elem` (map fst d))
post (Directory d) (Directory d') oldname newname =
  d' == take i d ++ [(newname, (map snd d)!!i)] ++ drop (i+1) d
  where i = elemIndex oldname (map fst d)

```

c) Spezifikation von `lookup` durch Haskell-Funktion

`hd` ist der erste Bestandteil des rekursiv abgebauten Pfadnamens, `tl` ist der Rest davon, beginnend mit `'/` (das durch "tail `tl`" abgeschnitten wird).

Bei `tl==""` ist man im Zielverzeichnis angekommen.

```

flookup :: File -> String -> File
flookup (Directory d) pname
  | tl==" " = dir
  | otherwise = flookup dir (tail tl)
  where (hd, tl) = break (=='/') pname
        Just dir = lookup hd d
flookup _ _ = error "not found"

```

d) Spezifikation von `list` durch Haskell-Funktion

```

list :: File -> IO ()
list (Directory d) = (putStr.unlines)
                    (map (\(a,b) -> a++"\t"++info b) d)
  where info (Text _) = "text"
        info (Program _) = "program"
        info (Directory _) = "directory"
list _ = error "can only list dirs"

```

e) Spezifikation von `create` durch Haskell-Funktion

`hd/tl` siehe oben

Bei `tl==""` ist man im Zielverzeichnis angekommen, falls der Name hier schon verwendet wird, Fehlermeldung.

`update` aktualisiert den durch den Pfadnamen beschriebenen Eintrag, andere bleiben unverändert.

```
create :: File -> String -> File -> File
create (Directory d) pname file
  | tl=="" = if hd `elem` (map fst d) then error "name clash"
              else Directory ((hd,file):d)
  | otherwise = Directory (map update d)
where (hd, tl) = break (=='/') pname
      update (k,v) = if k==hd then (k,create dir (tail tl) file)
                      else (k,v)
      Just dir = lookup hd d
```

Aufgabe 2 (Suchen in Bäumen)

Pseudo-Code:

- Steige in den Baum ab (Tiefensuche, aber Breitensuche geht auch) und füge für jeden Vorfahren mit dem passenden Namen eine Abstammungslinie zum Ergebnis hinzu.
- Beim Aufstieg (d.h. Rückkehr aus dem Abstieg) füge jeder Linie den aktuellen Knoten hinzu.

<http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U11.A2.zip>

Der Stammbaum eines Kindes wird durch den Konstruktor erzeugt. Dabei werden Verweise auf die Stammbäume der beiden Elternteile übernommen. Es wird also kein vollständiger neuer Baum erzeugt, sondern die zwei bestehenden werden als Bestandteile des neuen "wieder verwendet", es findet ein "Sharing" statt. Dies entspricht dem üblichen objektorientierten Paradigma, während in der funktionalen Programmierung der Stammbaum des Kindes ein neuer "unabhängiger" Wert wäre.

Das Sharing birgt immer dann Gefahren, wenn die gemeinsam genutzten Objekte modifizierbar sind. Durch Änderung eines Objekte ändert sich der Zustand aller anderen Objekte, die darauf Bezug nehmen. Dies ist im Fall der Stammbäume ausgeschlossen, da alle Attribute als final markiert sind, es handelt sich um konstante Objekte, was zur Semantik eines Stammbaums passt.

```

3 import java.util.Vector;
4
5 class NoSuchPerson extends Exception {}
6
7 public class Person {
8
9     private final Person father, mother;
10    private final String name;
11
12    Person(Person father, Person mother, String name) {
13        this.father = father;
14        this.mother = mother;
15        this.name = name;
16    }
17
18    public Vector<Vector<Person>> ancestors(String name) throws NoSuchPerson {
19
20        Vector<Vector<Person>> result = new Vector<Vector<Person>>();
21
22        // Test person itself
23        if (name.equals(this.name))
24            result.add(new Vector<Person>());
25
26        // Test father
27        if (father != null) {
28            try {
29                Vector<Vector<Person>> fatherAncestors = father.ancestors(name);
30                result.addAll(fatherAncestors);
31            } catch (NoSuchPerson e) {}
32        }
33
34        // Test mother
35        if (mother != null) {
36            try {
37                Vector<Vector<Person>> motherAncestors = mother.ancestors(name);
38                result.addAll(motherAncestors);
39            } catch (NoSuchPerson e) {}
40        }
41
42        if (result.size() == 0)
43            throw new NoSuchPerson();
44
45        for (Vector<Person> ancestorLines : result)
46            ancestorLines.add(this);
47
48        return result;
49    }
50 }

```

Aufgabe 3 (Praktische Javakenntnisse 3 - Protokollierung)

Die Lösung folgt dem Beispiel aus der Aufgabenstellung.

<http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U11.A3.zip>