

Übungsblatt 11 - Abgabe 17.01.05

Aufgabe 1 (Bäume als Datenspeicher)

Wir betrachten ein hypothetisches einfaches Dateisystem, das drei Arten von Dateien enthält: Textdateien, die Zeichenketten enthalten, Programmdateien zur Verarbeitung von Zeichenketten, ferner Verzeichnisse, die Zuordnungen von Dateinamen zu Dateien enthalten. Das Dateisystem wird wie folgt modelliert:

```
data File = Text String |
           Program(String->String) |
           Directory[(String, File)]
```

- Geben Sie eine möglichst gute Invariante an, formuliert als Boole'sche Haskell-Funktion! Falls Sie verschiedene Alternativen sehen, diskutieren Sie diese!
- Wir betrachten eine Operation `rename`, die in einem vorgegebenen Verzeichnis eine Datei umbenennt. Spezifizieren Sie diese Operation durch Angabe von Voraussetzung und Effekt, beides als Boole'sche Haskell-Funktionen.
- Wir betrachten eine Operation `lookup`, die zu einem vorgegebenen Pfadnamen `p` und zu einem vorgegebenen Verzeichnis `d` diejenige Datei liefert, die von `d` aus über `p` erreichbar ist. Diese Operation soll durch Angabe einer Haskell-Funktion `lookup` spezifiziert werden. Sehen Sie für die möglichen Ausnahmefälle geeignete `error`-Meldungen vor! (Zur Erinnerung: Ein Pfadname besteht (in Unix) aus einer Folge von Namen, die durch `'/'` getrennt sind, z.B. `"a/b/c"`)
- Wir betrachten eine Operation `list`, die den Inhalt eines gegebenen Verzeichnisses liefert – als Text mit geeignetem Layout und mit Angabe der Dateiarten, z.B.

```
letter1    text
myprog     program
letter2    text
library    directory
```

Diese Operation soll ebenfalls durch Angabe einer Haskell-Funktion spezifiziert werden.

- Wir betrachten eine Operation `create`, die eine „neue“ Datei mit vorgegebenem Pfadnamen in korrekter Weise in ein Dateisystem mit Wurzel `d` einträgt. Auch diese Operation soll durch Angabe einer Haskell-Funktion spezifiziert werden.

Aufgabe 2 (Suchen in Bäumen)

Der Stammbaum einer Person – ein binärer Baum – soll als Geflecht von konstanten Objekten (immutable objects) einer Klasse Person repräsentiert werden.

a) Geben Sie eine geeignete Klasse an. Sehen Sie einen Konstruktor vor, der es ermöglicht, den Stammbaum eines Kindes unter Verwendung der Stammbäume der Eltern zu erzeugen. Diskutieren Sie die dabei entstehende Datenstruktur (Stichwort Objekt-Sharing).

b) Entwickeln Sie für die ausgewählte Datenstruktur eine Methode `ancestors(String)`, welche ausgehend von einer Person alle Vorfahren mit dem übergebenen Namen findet und diese mit der Vererbungskette bis zur Person selbst zurückgibt (Reihenfolge: Vom Vorfahren bis zur Person selbst).

```
public Vector<Vector<Person>> ancestors(String
name)
                                throws NoSuchPerson
```

Aufgabe 3 (Praktische Javakenntnisse 3 - Protokollierung)

Die Protokollierung (Logging) der Tätigkeiten einer Applikation ist im Einsatz oft die einzige Möglichkeit Programmfehler zu finden. Wir betrachten für diese Übung das Protokollierungsrahmenwerk Log4j, welches Ihr unter <http://logging.apache.org/log4j/docs/> finden könnt. Die Architektur sowie Implementierung von Log4j ist entsprechend den laufzeitkritischen Anforderungen von Applikationen optimiert worden.

Log4j besteht aus drei wesentlichen Komponenten:

1. Logger

Ein Logger ist ein Objekt, an das die auszugebenden Meldungen übergeben werden. Logger können hierarchisch angeordnet werden, was in folgendem Beispiel demonstriert wird:

```
Logger l1 = Logger.getLogger("alp3");
Logger l2 = Logger.getLogger("alp3.bank.logging");
```

Für auszugebende Meldungen existieren die folgenden Prioritätsstufen:

```
DEBUG < INFO < WARN < ERROR < FATAL.
```

Jedem Logger kann ein solcher Level zugeordnet werden. Geschieht dies nicht, "erbt" er den Level seines Vorfahren. Eine Meldung kann nun durch Aufruf von `logger.debug(...)`, `logger.info(...)`, etc. an den Logger übermittelt werden und wird von diesem weitergeleitet, falls der Logger mindestens die Prioritätsstufe der Nachricht besitzt.

2. Appender

An einen Logger können mehrere Appender angeschlossen werden, welche dessen Meldungen auf verschiedene Arten ausgeben.

Die wichtigsten Appender für die praktische Arbeit sind der ConsoleAppender und FileAppender.

3. Layout

Bevor eine Meldung vom Appender ausgegeben wird, wird sie gemäß eines Layouts formatiert. Diese Layouts können den Appendern über eine Konfigurationsdatei zugeordnet werden.

Es stehen folgende Appender zur Verfügung:

SimpleLayout, PatternLayout, HTMLLayout, XMLLayout

Beispiel:

```
// Ein Logger-Objekt namens 'wecker' wird erzeugt
Logger logger = Logger.getLogger("wecker");

// ein Appender mit einem Layout wird erstellt
ConsoleAppender consoleAppender = new ConsoleAppender(new SimpleLayout());
// und dem Logger zugewiesen
logger.addAppender( consoleAppender );
// noch ein Appender
FileAppender fileAppender =
    new FileAppender( layout, "MeineLogDatei.log", false );
// der wird auch dem Logger zugewiesen
logger.addAppender( fileAppender );
// ALL | DEBUG | INFO | WARN | ERROR | FATAL | OFF:
logger.setLevel(Level.ERROR);
logger.debug("Jetzt wird geweckt.");
logger.info("Guten Morgen.");
logger.warn("Du solltest langsam aufstehen !");
logger.error("Jetzt kommst Du schon zuspät !");
logger.fatal("Na toll, nun hast Du die ALP3-Vorlesung verpennt.");
```

Eine deutsche Anleitung findet ihr unter:

<http://www.javacore.de/jumpToDownload.php?id=8&url=tutorials/schnelle/log4jmanual.pdf>

Aufgabe:

Als praktische Aufgabe solltet ihr die Protokollierung der Bank

(<http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U6.A1.jar>) auf Log4j umstellen. Protokolliert werden soll sowohl in eine Datei, als auch auf die Konsole.

Abgabe:

- Modifizierte Quellcodefragmente mit Angabe der zugehörigen Datei auf Papier.
- Vollständiger Quellcode und ausführbare JAR-Datei per Mail an den Tutor.