Praktische Informatik 3 WS 04/05

5. Übungsblatt

Ausgabe: 10.01.05
Abgabe: 25/26.01.05

Christoph Lüth <cxl>
Shi Hui <shi>
Klaus Lüttich <luettich>
Wolfgang Machert <wmachert>
Christian Maeder <maeder>
Hennes Märtins <maertins>
Kai-Florian Richter <richter>
Dennis Walter <dw>
Diedrich Wolter <dwolter>

10 *Wunder der Pflanzenwelt* *8 Punkte*

In der Informatik wachsen Bäume meist nach unten, in der Natur vielfach nach oben. Auch sonst gibt es viele Unterschiede: in der freien Natur haben Bäume meistens mehr als zwei Äste pro Stamm.

In dieser Aufgabe sollen sogenannte n-äre Bäume implementiert werden. Ein n-ärer Baum ist ein Baum, der pro Knoten eine beliebige Anzahl von Nachfolgern hat.

1. Zuerst geben Sie einen Datentypen `NTree a` an, der einen nicht-leeren n-ären Baum modelliert.

2. Mit einem Datentyp allein kann man noch nicht so viel anfangen, deshalb benötigen wir noch die folgenden Funktionen:

   ```
   fold   :: (a-> [b]-> b)-> NTree a-> b
   map    :: (a-> b)-> NTree a-> NTree b
   filter :: (a-> Bool)-> NTree a-> [NTree a]
   ```

   `filter` soll alle Knoten aus dem Baum entfernen, die nicht das Prädikat erfüllen. Dabei werden alle Unterbäume in den darüberliegenden Knoten kollabiert. Wenn also ein Knoten gelöscht wird, ist das Ergebnis die Liste seiner Unterbäume, ansonsten die ein-elementige Liste mit diesem Baum.

3. Implementieren sie folgende Hilfsfunktionen, die einen n-ären Baum traversieren:

   ```
   preorder  :: NTree a-> [a]
   postorder :: NTree a-> [a]
   ```

4. Ein *Pfad* in einem n-ären Baum ist die Liste der Knoten, die man von der Wurzel durchlaufen muss, um zu einem Knoten zu gelangen. Implementieren Sie eine Funktion

   ```
   paths :: NTree a-> [[a]]
   ```

   welche die Liste aller Pfade für alle Knoten in einem gegebenen Baum berechnet.

11 *Da ist ein Baum in meinem Rechner!* *6 Punkte*

n-äre Bäume sind weiter verbreitet, als man denkt: in den meisten Rechnern steckt mindestens einer, nämlich das Dateisystem.

In Haskell können wir eine einzelne Datei durch eine Datentyp repräsentieren, der den Namen der Datei, die Zugriffsrechte und die Zeit der letzten Modifikation enthält:

```
import Directory
import Time (ClockTime)

data File = File String Permissions ClockTime
```

Damit ist ein Dateisystem (oder genauer gesagt, eine Verzeichnisstruktur) ein n-ärer Baum von Dateien, und wir können jetzt eine Funktion implementieren, die eine Verzeichnisstruktur einliest:

```
readDir  :: FilePath-> IO (NTree File)
```

Die benötigten Funktionen zum Lesen von Verzeichnissen und so weiter finden Sie im Haskell98-Modul `Directory`. Beachten Sie die Fehlerbehandlung, die Funktion sollte nicht undefiniert abbrechen, wenn ein Fehler auftritt, weil Dateien oder Verzeichnisse nicht lesbar sind etc.

12 *Anzeigen, Suchen und Finden.* *6 Punkte*

Mit diesen Funktionen ist die Implementation des "*Haskell Tool Set*" (einer völlig in Haskell implementierten Sammlung von nützlichen Hilfsprogrammen) ein Kinderspiel. Wir werden hier prototypisch zwei dieser Hilfsprogramme implementieren, nämlich `ls` und `find`.

Beide sollen aus der Kommandozeile benutzt werden können, d.h. sie müssen mit den Funktionen aus dem Modul `System` die Argumente der Kommandozeile auslesen.

1. `ls` zeigt Verzeichnisse an, und sollte wie folgt benutzt werden können:

   ```
   ls [-l] [-R] files ...
   ```

   Hierbei soll die Option `-l` eine lange Ausgabe erzeugen (mit Angabe der Modifikationszeit und Rechten; ansonsten wird einfach nur der Name der Datei angezeigt), und `-R` zeigt rekursiv alle Unterverzeichnisse mit an (ansonsten nur die angegeben Verzeichnisse und Dateien).

   Die Ausgabe kann sich an dem Unix-Hilfsprogramm `ls` orientieren, muss aber nicht genauso aussehen.

2. `find` sucht nach Dateien in einem Verzeichnisbaum, die durch *Muster* spezifiziert werden. Ein Muster ist eine Folge von Zeichen, die wie folgt auf eine andere Folge von Zeichenketten passen kann oder nicht:

   - `?`, passt auf genau ein Zeichen;
   - `*`, passt auf beliebig viele Zeichen;

- `\*` und `\?` passen genau auf `?` und `*`;
- jedes andere Zeichen passt genau auf sich selbst.

Also zum Beispiel passt `"L*.hs"` auf `"List.hs"`, aber nicht auf `"Mist.hs"`; `"*.hs*"` passt auf `"List.hs"` sowie auf `"List.hs.gz"`.

Implementieren Sie zuerst eine Funktion

```
type Pattern = String

match :: Pattern-> String-> Bool
```

die prüft, ob das Muster auf die gegebene Zeichenkette passt.

Damit implementieren Sie dann eine Funktion, welche in einer durch `readDir` eingelesenen Verzeichnisstruktur nach Dateien sucht, auf deren Name das Muster passt.

Wir implementieren hier eine etwas einfachere Version des gleichnamigen Unix-Werkzeugs. Unser `find` sollte wie folgt benutzt werden:

```
find filepath -name pattern -name pattern ...
```

Hierbei ist `filepath` der Pfad, ab dem gesucht wird. Die `pattern` sind die Muster, nach denen gesucht werden soll; es soll mindestens eines der angebenen Muster passen.

Den Konventionen folgend sollen Fehlermeldungen auf der Standardfehlerausgabe (`stderr`) ausgegeben werden.

Hinweise:

1. Folgende Standardmodule könnten hilfreich sein:
   - `System`: Kommandozeilenargumente lesen (`getArgs`) und Programmname lesen (`getProgName`);
   - `IO`: Handle für Standardfehlerausgabe `stderr`, Ausgabe auf einen `Handle` (`hPutStrLn`);
   - `ClockTime`: Repräsentation der Systemzeit (`ClockTime`), Konversion in Ortszeit (`CalendarTime`) und Ausgabefunktionen (`calendarTimeToString`, `formatCalendarTime`).
2. Beachten Sie beim Testen von `find`, dass u.U. die Benutzerkommandozeilenschnittstelle (die *shell*) Muster mit `*` und `?` expandiert, bevor Ihr Programm sie überhaupt zu sehen bekommt. Abhilfe schaffen hier meistens die einfachen Anführungszeichen (z.B. `'*.hs'`).
3. Um ein Programm aus der Kommandozeile benutzen zu können, muss es entweder (mit dem `ghc`) übersetzt werden, oder mit `runhugs` gestartet werden.

Dies ist Version 1.1 vom 2005/01/11 12:46:33.