Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Lars Hupel, Dr. Jasmin Blanchette

WS 2013/14
07.01.2014
Abgabe: 14.01.2014, 15:30

Einführung in die Informatik 2
11. Übung

Aufgabe G11.1 Maps I: Assoziationslisten

Wir haben bereits in Aufgabe G2.2 Assoziationslisten `Eq k => [(k,v)]` als eine Möglichkeit kennengelernt, Maps mit Schlüsseltyp k und Werttyp v darzustellen.

Um zu verhindern, dass der Benutzer unsinnige Assoziationslisten erstellen kann (z.B. mit mehreren Einträgen für einen Schlüssel), wollen wir die Implementation in einem Modul verstecken.

Definieren Sie ein Modul `AssocList`, das einen Typ `Map k v` und die folgenden Funktionen bereitstellt. Andere Typen und Funktionen sollen nicht exportiert werden! Insbesondere sollen die Konstruktoren von `Map` nicht zur Verfügung stehen.

```
newtype Map k v = ...
empty :: Eq k => Map k v
insert :: Eq k => k -> v -> Map k v -> Map k v
lookup :: Eq k => k -> Map k v -> Maybe v
delete :: Eq k => k -> Map k v -> Map k v
keys :: Eq k => Map k v -> [k]
```

(Bei einem Aufruf von `insert` mit bereits existierendem Schlüssel soll der alte Wert überschrieben werden.)

Die Maps sollen intern als Assoziationslisten dargestellt werden.

Hinweis: Die Prelude bietet auch eine `lookup`-Funktion an. Um Namenskonflikte zu vermeiden, können Sie mittels `import Prelude hiding (lookup)` die vordefinierte Funktion in Ihrem Modul verstecken.

Aufgabe G11.2 Na warte, Schlange!

Eine *Warteschlange* ist eine klassische Datenstruktur mit FIFO-Semantik ("first in, first out"). Neue Elemente werden am Ende der Schlange hinzugefügt, alte Elemente werden am Kopf entfernt. Schreiben Sie ein Modul, dass einen passenden Datentyp `Queue` mit geeigneten Operationen definiert. Die Implementationsdetails sollen versteckt werden.

Aufgabe G11.3 Die Tippfehler schlagen zurück (optional)

Der Datentyp

```
data Either a b = Left a | Right b
```

ist neben `Maybe` eine der häufig verwendeten Möglichkeiten, Fehlerzustände zu signalisieren. Es ist Konvention, dass der `Left`-Konstruktor für den Fehlerwert und der `Right`-Konstruktor für einen korrekten Rückgabewert verwendet wird (Eselsbrücke: *right* ↔ *richtig*).

In Aufgabe H7.3 sollten Sie eine Funktion

```
fixTypos :: [String] -> [String] -> [String]
```

die eine Vokabelliste *vocabs* und eine Liste von teilweise falsch geschriebenen Wörtern *text* nimmt. Der Aufruf `fixTypos` *vocabs text* gibt eine Liste korrigierter Wörter (den *korrigierten Text*) zurück. Ein Wort ist *falsch geschrieben*, wenn es nicht in der Vokabelliste *vocabs* steht. Zwei Wörter sind *quasi-identisch*, wenn sie die gleiche Länge haben und das eine Wort aus dem anderen hergeleitet werden kann, indem genau ein Buchstabe durch einen anderen ersetzt wird. Zur Korrektur eines falsch geschriebenen Wortes wird dieses durch ein in *vocabs* vorhandenes quasi-identisches Wort ersetzt. Sind keine oder mehrere unterschiedliche quasi-identische Wörter vorhanden, soll das falsche Wort unverändert bleiben.

Dies Funktion hat das Problem, dass man dem Ergebnis nicht ansehen kann, welche Wörter korrekt waren, welche korrigiert und welche nicht korrigiert wurden, weil kein passendes Wort (oder mehr als ein passendes Wort) im Wörterbuch vorkommt.

Schreiben Sie daher eine Funktion

```
safeFixTypos :: [String] -> [String]
    -> [Either (String, [String]) String]
```

die einen korrigierten Text wie folgt zurückliefert: Ist ein Wort w falsch geschrieben, so wird es durch `Left (`w`,` vs`)` ersetzt, wobei vs die Liste der in *vocabs* vorkommenden Wörter ist, die quasi-identisch zu w sind. Andernfalls wird w durch `Right` w ersetzt.

Bauen Sie gegebenfalls auf die Musterlösung für H7.3 auf.

Beispiele:

```
vocabs = ["i", "more", "more", "now", "want", "won", "wow"]

safeFixTypos vocabs ["i", "want", "mose"] ==
  [Right "i", Right "want", Left ("mose", ["more"])]
safeFixTypos vocabs ["and", "j", "want", "it", "naw"] ==
  [ Left ("and", []), Left ("j", ["i"]), Right "want"
  , Left ("it",[]), Left ("naw", ["now"])]
safeFixTypos vocabs ["won", "", "now", "wow", "wob", "nob"] ==
  [ Right "won", Left ("", []), Right "now", Right "wow"
  , Left ("wob", ["won", "wow"]), Left ("nob", ["now"])]
safeFixTypos vocabs ["rome", "boban", "wat", "ant", "watn"] ==
  [ Left ("rome", []), Left ("boban", []), Left ("wat", [])
  , Left ("ant",[]), Left ("watn", [])]
```

Wichtig: Ihre Übungsabgabe besteht aus mehreren Dateien. Laden Sie zusätzlich zur `Exercise_11.hs` auch noch `Map2.hs` und Ihre veränderte `AssocList.hs` (siehe G11.1) in das Verzeichnis `exercise_11` hoch.

Aufgabe H11.1 Maps II (9 Punkte)

In Aufgabe G11.1 haben Sie bereits ein Modul für Maps implementiert.

1. Schreiben Sie ein weiteres Modul `Map2`, dass einen Typ `Map k v` und die folgenden Funktionen bereitstellt. Andere Typen und Funktionen sollen nicht exportiert werden! Insbesondere sollen die Konstruktoren von `Map` nicht zur Verfügung stehen.

   ```
   empty :: Ord k => Map k v
   insert :: Ord k => k -> v -> Map k v -> Map k v
   lookup :: Ord k => k -> Map k v -> Maybe v
   delete :: Ord k => k -> Map k v -> Map k v
   keys :: Ord k => Map k v -> [k]
   mapV :: Ord k => (v -> w) -> Map k v -> Map k w
   ```

   Überlegen Sie sich zur Darstellung der Map eine geeignete Datenstruktur (z.B. eine Art Baum). Die Darstellung als Assoziationsliste oder ähnliche Datenstruktur sowie die Verwendung von `Data.Map` und vergleichbaren fertigen Strukturen ist nicht zulässig.

   Für die automatisierten Tests achten Sie bitte besonders auf die korrekte Benennung der Datei (Groß- und Kleinschreibung ist wichtig)!

2. Schreiben Sie in der `Exercise_11.hs` Funktionen `assocListToMap2` and `map2ToAssocList`, um zwischen den beiden Map-Typen aus `AssocList` und `Map2` zu konvertieren. Welche Typsignatur brauchen diese Funktionen?

3. Die Implementierung von `assocListToMap2` und `map2ToAssocList` ist nicht effizient, da sie zuviel Arbeit verrichtet. Warum?

   Ergänzen Sie die Module `Map2` und `AssocList` so, dass eine effizientere Implementierung möglich ist. Wenn Sie zusätzliche Funktionen exportieren, kennzeichnen Sie diese. Dabei soll die interne Repräsentation weiterhin geheimgehalten werden und keines der beiden Module soll von dem jeweils anderen Modul abhängen.

Aufgabe H11.2 Timeo daemones et dies ferentes. (6 Punkte)

In dieser Aufgabe implementieren Sie einen Netzwerkserver, der Clients auf Wunsch die Uhrzeit mitteilt.

Die Kommunikation mit dem Server verläuft streng nach dem I2B11H2P (Info 2, Blatt 11, Hausaufgabe 2-Protokoll):

1. Der Server wartet auf dem Port 9002 auf eine Verbindung.

2. Ein Client verbindet sich mit dem Server.
3. Bei Verbindungsaufbau meldet sich Ihr Server mit `I2B11H2P/1.0 HELLO`, gefolgt von einem Zeilenumbruch.
4. Der Client sendet daraufhin entweder `DATE` oder `TIME` (jeweils gefolgt von Whitespaces), woraufhin der Server mit dem aktuellem Datum (im Format `YYYY-MM-DD`) bzw. der aktuellen Uhrzeit (im Format `HH:MM:SS`) antwortet (jeweils gefolgt von einem Zeilenumbruch).

   Beispiel 1: `"DATE\n"` – `"2014-01-01\n"`

   Beispiel 2: `"TIME\n"` – `"00:00:01\n"`
5. Der Server terminiert die Verbindung und wartet erneut auf eine Verbindung, so dass das Protokoll wieder von vorn beginnt.

Für die volle Punktzahl muss Ihr Programm so implementiert sein, dass ein sich an das Protokoll haltender Client stets die gewünschten Angaben erhält.

Beachten Sie, dass der Server nach einem Protokollablauf oder einer fehlerhaften Eingabe nicht einfach beendet wird. Bei einer fehlerhaften Eingabe darf die aktuelle Verbindung jedoch getrennt werden.

Definieren Sie eine Funktion `server :: IO ()`, die einen Server startet und das Protokoll initiiert. Diese Funktion soll nicht terminieren. Die dafür nötigen Funktionsaufrufe finden Sie in der Übungsschablone.

Hinweis: Nutzen Sie z.B. Hoogle, um sich über Funktionen wie `hPutStrLn` zu informieren.

Aufgabe W12.1 Othello (zweiwöchige Wettbewerbsaufgabe, 0 Punkte)

Diese vorletzte, zweiwöchige Aufgabe besteht darin, künstliche Intelligenz (KI) für das Spiel *Othello* zu programmieren. Der Master of Competition hat die Idee von seinem Kollegen Dr. Tjark Weber übernommen. Teile der Erklärungen unten sind mit leichten textuellen Änderungen dem Wikipedia-Artikel über das Spiel[1] entnommen.

Othello ist ein Brettspiel für zwei Personen, die *schwarze* bzw. *weiße* Spielsteine benutzen. Das Brett besteht aus 8×8 Feldern. Jedes Feld kann entweder frei oder mit einem (schwarzen oder weißen) Stein besetzt sein. Die Felder sind als (*Zeile*, *Spalte*) indiziert, wobei die Nummerierung bei 0 beginnt. Zu Beginn des Spiels sind die vier mittleren Felder besetzt: $(3, 3)$ und $(4, 4)$ sind weiß; $(4, 3)$ und $(3, 4)$ sind schwarz. Alle anderen Felder sind frei.

Die Spieler sind abwechselnd an der Reihe. Schwarz fängt an. Ein Zug besteht darin, dass der Spieler einen Stein seiner Farbe auf ein freies Feld setzt. Der Spieler darf nur so setzen, dass ausgehend von dem gesetzten Stein in beliebiger Richtung (senkrecht, waagerecht oder diagonal) ein oder mehrere gegnerische Steine anschließen und danach wieder ein eigener Stein liegt. Es muss also mindestens ein gegnerischer Stein von dem gesetzten Stein und einem anderen eigenen Stein in gerader Linie eingeschlossen werden. Dabei müssen alle Felder zwischen den beiden eigenen Steinen von gegnerischen Steinen besetzt sein. Alle gegnerischen Steine, die so eingeschlossen werden, wechseln die Farbe, indem sie umgedreht werden. Ein Zug kann mehrere Reihen gegnerischer Steine gleichzeitig einschließen, die dann alle umgedreht werden.

Der Spieler, der an der Reihe ist, muss einen Stein setzen. Hat er keine Möglichkeit, einen Stein regelkonform zu setzen, muss er passen. Müssen beide Spieler unmittelbar nacheinander passen (z.B. wenn das Brett voll ist), ist das Spiel beendet. Der Spieler, der am Ende die meisten Steine seiner Farbe auf dem Brett hat, gewinnt. Bei gleicher Zahl ist das Spiel unentschieden.

Jetzt konkreter zu Ihrer Aufgabe. Sie besteht darin, eine Datei namens `Othello_Player.hs` einzureichen. Diese muss auf dem folgenden Modul basieren:

```
module Othello_Base where

data Color = Black | White
  deriving (Eq, Show)

data Move = Pass | Play (Int, Int)
  deriving (Eq, Show)
```

Ihr Modul `Othello_Player` sollte wie folgt aussehen:

```
{-WETT
  ...
-}

module Othello_Player (State, startState, nextState) where
```

[1] `http://de.wikipedia.org/wiki/Othello_(Spiel)`

```
import Othello_Base

data State = ...

startState :: Color -> State
startState whoAmI = ...

nextState :: State -> Move -> (Move, State)
nextState opponentMove oldState = ...
```

Der Typ `State` stellt den internen Zustand Ihres Programmes dar.

Die Funktion `startState` nimmt die Farbe des KI-Spielers als Argument und gibt den Startzustand zurück.

`nextState` nimmt zwei Argumente: der alte Zustand und der aktuelle Zug des Gegners. Am Spielanfang ist der „aktuelle“ Zug `Pass`. Die Funktion gibt ein Paar (*Zug*, *Zustand*) zurück, in dem *Zug* der von der KI gewünschte Zug und *Zustand* der neue Zustand nach diesem Zug (und nach dem vorherigen Zug des Gegners) darstellt. Dieser Zustand wird (zusammen mit der Antwort des Gegners auf *Zug*) im nächsten `nextState`-Aufruf als Argument angegeben. *Zug* darf dann und nur dann `Pass` sein, wenn kein Stein gesetzt werden kann.

Falls die KI schwarz spielt, sieht die Sequenz der Aufrufe wie folgt aus:

```
let state0 = startState Black
let (myMove1, state1) = nextState state0 Pass
let (myMove2, state2) = nextState state1 opponentMove1
let (myMove3, state3) = nextState state2 opponentMove2
...
```

Falls die KI weiß spielt, sieht die Sequenz der Aufrufe wie folgt aus:

```
let state0 = startState White
let (myMove1, state1) = nextState state0 opponentMove1
let (myMove2, state2) = nextState state1 opponentMove2
...
```

Ganz oben in der `Othello_Player.hs`-Datei sollte ein Kommentar der Form

```
{-WETT
  ...
-}
```

stehen. Dieser sollte Ihre Implementierung beschreiben und relevante Quellenangaben enthalten.

Um Ihre Implementierung zu testen, können Sie den einfachen Treiber `Othello_Driver` nutzen (siehe Übungswebseite). Er setzt voraus, dass `State` die Typklasse `Show` unterstützt.

Die (Co(ntra)-)MCs werden ein Turnier mit allen eingereichten KI-Programmen organisieren. Jede KI wird gegen jede KI spielen, einmal als schwarz und einmal als weiß. Falls Ihre Implementierung einen falschen Zug zurückgibt oder sonst Fehler aufweist, werden Sie disqualifiziert.

Für jedes Spiel wird Ihre KI eine Punktzahl zwischen $+64$ und -64 erhalten:

$$\textit{Punktzahl} = \langle \textit{Anzahl Felder Ihrer Farbe} \rangle - \langle \textit{Anzahl Felder der anderen Farbe} \rangle$$

Jedem Programm werden fünf Minuten pro Spiel zur Verfügung gestellt. Es darf keine neuen Threads oder Prozesse starten. Bei Timeout oder anderen technischen Problemen bekommt der Gegner $+64$ und Sie -64 Punkte. Die Hardware des MCs ist fantastisch, trotzdem sollten Sie es nicht übertreiben.

Hinweis 1: Es existiert eine Fülle von Verfahren, um dieses Problem zu lösen. Statt das Rad neu zu erfinden, könnten Sie überlegen, einen etablierten Ansatz zu implementieren (Stichwort *Computer Othello*[2]). In diesem Fall sollten Sie nicht vergessen, Ihre Lösung mit einer Quellenangabe zu vorsehen. Es wird aber erwartet, dass Sie den Code selbst (eventuell als Mitglied eines Teams) geschrieben haben.

Hinweis 2: Dr. Weber rät: Do not go overboard. We understand that one could spend a lifetime perfecting an Othello AI, while you only have a few days.

Wichtig 1: Diese Wettbewerbsaufgabe ist am 21.01.2014 als Teil von Blatt 12 abzugeben. Sie bringt bis zu 60 Wettbewerbspunkte. Ab ca. 13.01.2014 werden wir erste Vorab-Turniere durchführen und Sie über Ihren aktuellen Stand im Ranking informieren.

Wichtig 2: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken.

[2] z. B. `http://en.wikipedia.org/wiki/Computer_Othello`