Universität Bremen

Praktische Informatik 3 WS 22/23

Christoph Lüth
Raphael Baass
Thomas Barkowsky
Tede von Knorre
Alexander Krug
Tarek Soliman

8. Übungsblatt

Ausgabe: 19.12.2022
Abgabe: 15.01.2023

Genug von Graphen, Tabellen und den sonstigen wilden Ideen ihres Bekannten. Es wird Zeit, die eigene Karriere voranzutreiben. Als aufstrebender Haskell-Entwickler wollen sie in der Spielebranche Fuß fassen. Um ihr Resümee aufzupolieren, wollen sie ihre eigene Version des Klassikers *Minesweeper* in Haskell implementieren. (Falls Sie Minesweeper nicht kennen sollten, bietet wie immer Wikipedia[1] einen informativen Überblick.)

Dazu müssen wir zuerst das Spielfeld modellieren. Eine einzelne Zelle wird repräsentiert durch

```
data Cell = Field | Bomb | Open Int | Flag Bool deriving Eq
```

Hierbei steht `Field` für ein leeres Feld, `Bomb` für ein Feld mit Bombe, `Open` für ein angeschautes Feld mit der Zahl der benachbarten Bomben und `Flag` für eine Flagge, mit einem Boolean der angibt, ob unter der Flagge eine Bombe ist (`True`) oder nicht (`False`).

Damit konstruieren wir einen Datentypen, der das ganze Spielfeld modelliert:

```
data Board = Board { cells :: M.Map (Int,Int) Cell }
```

Das Spielfeld ist also eine endliche Abbildung von Paaren (x,y) auf einzelne Zellen.

8.1 *Minen legen* — *3 Punkte*

Wir fangen mit drei Funktionen an, die zusammen ein neues Spielfeld erzeugen. Zuerst erzeugen wir ein mit leeren Feldern gefülltes Spielfeld (wobei die beiden Parameter die Breite und Höhe des Spielfeldes angeben; wir zählen jeweils von 1 bis b bzw. h, wenn b und h Breite und Höhe sind):

```
newBoard :: Int → Int → Board
```

Die folgende Funktion versieht ein Feld mit Minen an den angegebenen Positionen:

```
fillBoard :: Board → [(Int,Int)] → Board
```

Jetzt müssen wir nur noch zufällig die Koordinaten der Bomben berechnen. Dazu implementieren wir eine folgende Funktion:

```
randomPositions :: Int → Int → Int → IO [(Int,Int)]
```

Die Parameter sind hier wieder die Breite und Höhe des Feldes, und die Anzahl der gewünschten Bomben. Dabei müssen wir darauf achten, dass in den zufällig generierten Koordinaten keine Duplikate vorkommen und trotzdem genug Bomben eingefügt werden.

Hinweise:

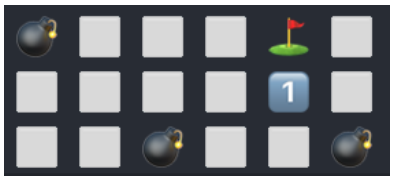
1. Konsultiert die (umfangreiche!) Dokumentation für `Data.Map`[2], um das `Board` zu manipulieren. Hilfreich sind besonders die Funktionen `empty`, `lookup`, `!`, `insert`, `adjust`, `fromList`.
2. Zufallszahlen können generiert werden mit

   `reandomRIO(0,5)` -- generiert eine zufällige Zahl zwischen 0 und 5

8.2 *Züge* — *8 Punkte*

Die folgende Funktion implementiert einen Spielzug (d.h. das Aufdecken an einer Position):

[1] `https://de.wikipedia.org/wiki/Minesweeper`
[2] `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html`



```
X . . . F .
. . . . 1 .
. . X . . X
```

Abbildung 1: Ein Beispiel-Spielfeld (links die Unicode-Darstellung, rechts einfaches ASCII.

```
move :: Board → Int → Int → Maybe Board
```

Die Parameter sind das bisherige Board, sowie die Koordinaten, die aufgedeckt werden sollen. Wenn an den gegebenen Koordinaten eine Bombe war, wird `Nothing` zurückgegeben. Ansonsten soll das Feld an den gegebenen Koordinaten in angeschaut (`Open`) geändert werden, mit der errechneten Anzahl der benachbarten Bomben. Wenn die Koordinaten nicht legal sind, soll das unveränderte Spielfeld gegeben werden.

Die Anzahl der benachbarten Bomben berechnet die Funktion

```
calculate :: Board → Int → Int → Int
```

Hierbei werden die Felder über, unter, rechts und links, sowie diagonal von den Koordinaten als benachbart angesehen. Hierzu könnte es hilfreich sein eine Funktion zu implementieren, welche die Liste der benachbarten Felder berechnet, worüber man dann die Anzahl der Bomben zählen kann. Man beachte, dass die Felder am Rand natürlich weniger Nachbarn haben.

Sehr ähnlich zu `move` schreiben wir dann noch eine Funktion, welche an den angebenen Koordinaten eine Flagge plaziert:

```
flag :: Board → Int → Int → Board
```

Die Funktion `checkForWin` soll schließlich prüfen, ob eine Gewinnstellung erreicht wurde:

```
checkForWin :: Board → Bool
```

Eine Gewinnstellung wurde erreicht, wenn alle Felder entweder aufgedeckt, oder Bomben mit einer Flagge versehen wurden.

8.3 *Von und Zu String* *4 Punkte*

Nun wollen wir uns unser generiertes Spielfeld auch anschauen. Zunächst implementieren wir die Darstellung von Cell mit:

```
instance Show Cell where
    show...
```

Dabei soll das Feld durch den Unicode Charakter `U+2B1C` dargestellt werden, eine Bombe durch '`U+1F4A3`', eine Flagge durch '`U+26F3`. Die aufgedeckten Felder sollen durch `"0\xFE0F\x20E3 "` dargestellt werden, wobei die 0 am Anfang der jeweiligen Zahl entspricht. Diese Kodierungen sind durch die Funktionen `open_str`, `bomb_str`, `flag_str` und `field_str` schon vorgegeben.[3] Für eine besser Formatierung fügen wir zwischen den Zellen Leerzeichen ein.

Damit implementieren wir eine Instanz von `Show` für das Spielfeld:

```
instance Show Board where
    show...
```

Die Ausgabe sieht dann so aus wie in Abbildung 1. Der Spieler soll natürlich das Spielfeld *ohne* Bomben sehen, sonst wäre der Spielspaß gering. Zu diesem Zweck implementieren wir eine Funktion, ähnlich der Show-Instanz:

```
displayBoard :: Board → String
```

Den Zustand des Spieles speichern und später wieder laden zu können wäre auch nicht schlecht. Um das Laden später realisieren zu können, brauchen wir eine Funktion, die ein Spielfeld in eine einfache String-Repräsentation und zurück umwandelt:

[3] Diese Unicode-Darstellung funktioniert leider nicht unter allen Umgebungen, bwpsw. nicht unter Windows (Power Shell) oder Linux/KDE. Das kann zu Laufzeitfehlern führen wie `<stdout>: commitBuffer: invalid argument (invalid character)!`
In diesem Fall ändern Sie bitte die Definition der Konstante `unicode` auf `False`.

```
boardFromString :: String → Board
boardToString   :: Board → String
```

Dabei soll ein `'X'` eine Bombe darstellen, `'-'` ein leeres Feld, `'0'` ein offenes Feld (die Zahl der benachbarten Bomben muss neu berechnet werden), `'f'` eine Flagge auf einem leeren Feld und `'F'` eine Flagge auf einer Bombe. Der Zeilenumbruch markiert eine neue Zeile. Das Spielfeld aus Abbildung 1, mit Höhe 3 und Breite 6, einer Flagge auf einer Bombe an Position (1,5) und einem offenen Feld an Position (2,5) wird kordiert als

```
boardToString b ⇝ "X---F-\n----0-\n--X--X"
```

8.4 *Kommunikation mit der Außenwelt* *5 Punkte*

Die Ein- und Ausgabe wird in dem Modul `Main` implementiert. Zuerst schreiben wir zwei Funktionen, mit denen wir Spielstände in Dateien schreiben und von dort wieder lesen können:

```
save :: Board → String → IO ()
load :: String → IO Board
```

Um die einen String in eine Datei zu schreiben bzw. von dort zu lesen, können Sie folgende vordefinierten Funktionen nutzen. Denken Sie daran, Fehler abzufangen!

```
writeFile :: FilePath → String → IO ()}(FilePath ist ein String).
readFile  :: FilePath → IO String
```

Jetzt wollen wir aber endlich spielen! Dazu müssen wir die Kommandos des Spieler verstehen. Die möglichen Kommandos des Spielers modellieren wir wie folgt:

```
data Command = Save String | Load String | SetFlag (Int, Int) | Move (Int, Int)
```

Um nun den Eingabestring des Spielers in die Kommandos zu parsen implementieren wir

```
parseMove :: String → Maybe Command
```

Dabei soll in einer an den `ghci`[4] angelehnten Syntax `:l spielstand_1` einen Spielstand aus der angegeben Datei laden, `:s spielstand_1` einen Spielstand speichern, `:m 1 1` das Feld an der Koordinate (1,1) aufdecken und `:f 1 1` an der Stelle (1,1) eine Flagge setzten. Wenn kein gültiges Kommando eingegeben wird, soll Nothing zurückgegeben werden. Sie können die Koordinaten ganz naiv mit *read* lesen.

Das interaktive Spiel implementieren wir mit einer Funktion

```
play :: Board → IO
```

In dieser Funktion geben wir das Spielfeld aus, und warten auf eine Eingabe des Spielers. Diese können wir mit `parseMove` in ein Kommando umwandeln, und dann die entsprechenden Funktion aufgerufen werden: Nach dem Speichern und dem Setzen eine Flagge soll einfach weiterspielt werden. Nach dem Laden wird mit dem geladenen Spielstand weitergespielt. Nach dem Aufdecken eines Feldes wird mit der Funktion `move` ein neuer Spielstand berechnet. Ist dieser undefiniert (`Nothing`), hat der Spieler eine Bombe aufgedeckt, und ist davon angemessen dramatisch zu informieren; das Programm wird beendet. Ansonsten wird mit `checkForWin` geprüft, ob eine Gewinnstellung erreicht wurde; ist dies der Fall, wird eine beliebig schmuckvoll gestaltete Rückmeldung an den Spieler ausgegeben, und das Programm beendet, ansonsten wird das Spiel mit dem neuen Spielstand fortgesetzt.

Zum Schluss machen wir aus dem Spiel ein ausführbares Programm, dem die Größe des Spielfeldes (Höhe und Breite), sowie die Anzahl der Bomben als Kommandozeilenparameter übergeben werden. Dazu implementieren wir eine Funktion

```
main :: IO ()
```

Diese liest mit der Funktion `getArgs` aus `System.Environment` die Kommandozeilenparameter. Um die Validierung der Parameter sowie der Anzahl dieser machen wir uns keine Sorgen, und wandeln die Parameter ganz naiv mit `read` in ganze Zahlen um.

Dann erzeugen wir ein Spielfeld mit den gegebenen Parametern mit den entsprechenden Funktionen aus Übung 8.2, und rufen die Funktion `play` mit dem initialen Spielstand auf.

Ausführen kann man das Programm entweder, indem man mit `stack build` eine Executable erstellt und ausführt, mit `stack run 5 5 10` das Programm direkt startet,oder indem im GHCi bei geladenem Modul `Main` das Kommando `:run main 5 5 10` eingibt, wobei die Zahlen die erwähnten Parameter sind.

[4] Oder den beliebten Editor `vi`