Minesweeper in Haskell

Im Rahmen der Veranstaltung „Weitere Programmiersprache“ wurde eine Implementation des bekannten Spiels *Minesweeper* in der funktionalen Sprache *Haskell* implementiert. Hierfür wurden mehrere Datentypen definiert und viele verschiedene rekursive Funktionen implementiert, um die Spiellogik von *Minesweeper* zu beschreiben. Die Oberfläche wurde nur auf der Kommandozeile umgesetzt.

Kurzbeschreibung: Minesweeper

Minesweeper ist ein weltweit bekanntes Spiel, bei welchem es darum geht, aus einer Vielzahl von Kästen alle freien Felder anzuklicken und dabei die Minen zu ignorieren bzw. zu markieren. Die Oberfläche von Minesweeper besteht typischerweise aus einem Grid bzw. einer Matrix von Kästen.

Feldertypen

Diese sind bei Spielbeginn noch zugedeckt und können erst durch Anklicken (bzw. über Koordinaten) aufgedeckt werden. Sollte sich unter dem angeklickten Feld eine Mine befinden, so endet sofort das Spiel mit einer Niederlage. Außer einer Mine kann das Feld auch leer sein oder eine Nummer kann sich darauf befinden. Diese zeigt die Anzahl an Minen auf den dazu adjazenten Feldern an. Anhand dieser Nummern kann der Spieler Schlussfolgerungen ziehen, wo sich eine Mine befinden kann.

Funktionen des Spiels

Das Spiel bietet zwei Funktionen an, die man auf die unaufgedeckten Felder anwenden kann. Das bereits erwähnte Aufdecken des Feldes sorgt dafür, dass sich der Inhalt des Feldes offenbart. Hierbei kann es selbstverständlich passieren, dass der Spieler eine Mine findet und somit das Spiel verliert. Die andere wichtige Funktion ist das Markieren der Felder. Diese Funktion ist dafür gedacht, dem Spieler die Möglichkeit zu geben Felder zu markieren, unter denen sich nach Ansicht des Spielers eine Mine befindet. Dies sorgt dafür, dass das Feld nicht mehr aufdeckbar ist, um ein versehentliches Klicken auf eine bereits erkannte Mine zu verhindern. Der Spieler muss im Laufe des Spiels alle Minen erkennen und markieren.

Sobald der Spieler alle Minen erkannt und markiert hat, endet das Spiel und der Spieler hat das Spiel gewonnen. Das Spiel kann durch mehrere Kriterien erschwert werden: die Feldgröße (bzw. Matrix-Größe) und die Anzahl an Minen im Feld.

Implementation des Spiels in Haskell

Um das Spiel zu implementieren, musste zunächst einige Datentypen definiert, welche zwar nicht zwingend notwendig sind, um das Spiel in Haskell zu implementieren, jedoch sinnvoll sind, da sie den Code übersichtlicher machen und damit unerwünschtes Verhalten reduzieren.

Organisatorisches

Es wurde Stack genutzt, um die Abhängigkeiten der Software zu managen und das Projekt zu builden (siehe ReadMe). Zur Versionsverwaltung wurde GIT genutzt unter der URL: *https://github.com/fadais/minesweeper*

Datentypen

```
--  a cell: contains its own index in the Field (x,y); Number of mines in adjacent
cells; visibilty flag; mark flag
type Cell = (Index, Integer, Bool, Bool)
```

Zunächst wurde ein Datentyp für eine einzelne Zelle implementiert. Dabei handelt es sich ein 4-Tupel mit folgendem Inhalt:

1. Index: Die **x- und y Koordinate** der Zelle innerhalb des Spielfelds. Diese wird genutzt, damit determiniert werden kann, zu welchen Zellen diese Zelle adjazent ist. Der Datentyp Index wurde auch implementiert und wird weiter unten erklärt. Diese Koordinaten, werden schon zu Beginn bei der Generierung des Spielfeldes erzeugt und ändern sich nicht.
2. Integer: Hierbei handelt es sich um den **Inhalt der Zelle**. Dieser kann entweder
    - leer sein, also den Integer-Wert *0* besitzen.
    - die Anzahl der Minen in adjazenten Zellen zur aktuell betrachteten Zelle besitzen. Diese Anzahl wird schon beim Spielbeginn anhand der zuvor zufällig generierten Minen ermittelt und fest in die Zellen gespeichert. Da es nur insgesamt maximal acht adjazente Zellen zu jeder Zelle gibt, kann dieser Wert also nur zwischen *1* und *8 variieren.*
    - eine Mine sein, wobei die Zelle dann den Wert *-1* besitzt
3. Bool: Der dritte Wert des Tupels ist die **visibilty-flag**. Dabei handelt es sich um einen Bool Wert (kann entweder True (Wahr) oder False (Falsch) sein), um zu speichern, ob eine Zelle bereits aufgedeckt wurde, somit also der Inhalt der Zelle ausgegeben werden kann. Standardmäßig wird dieser Wert bei der Generierung des Spielfeldes initial auf False gesetzt, da zu Beginn alle Felder nicht angezeigt werden sollen. Sobald der Spieler eine Zelle aufdeckt, wird der Wert zu True.
4. Bool: Der vierte Wert des Tupels ist die **mark-flag**. Ähnlich wie die visibilty-flag ist die mark-flag auch ein Bool-Wert, kann also auch nur True oder False sein. Diese flag ist dazu da, um zu speichern, ob die Zelle markiert wurde und somit als **#** ausgegeben wird. Markierte Zellen können nicht mehr aufgedeckt werden, es sei denn sie werden vorher wieder demarkiert. Initial wird dieser Wert bei jeder Zelle auf False gesetzt, da logischerweise zu Beginn noch keine Zelle markiert wurde.

```haskell
-- tupel containing both x and y coordinates; is used in cells
type Index = (Integer, Integer)
```

Der Index-Datentyp wird, wie bereits oben erwähnt, genutzt, um die x- und y- Koordinaten einer Zelle innerhalb des Spielfelds zu speichern und somit die Position der Zelle, sowie die dazu adjazent liegenden Zellen bestimmen zu können. Bei dem Datentypen handelt sich um einfaches Tupel zweier Integer-Werte, wo bei der erste die X-Koordinate und der zweite die Y-Koordinate repräsentiert.

```haskell
-- minesweeper field containing all cells
type Field = [Cell]
```

Der Field-Datentyp repräsentiert das gesamte Spielfeld. Es handelt sich dabei, um eine Liste an Zellen.

Konstanten

Es wurden verschiedene feste Werte, sogenannte Konstanten, in das Spiel implementiert, welche, ähnlich zu den Datentypen, für einen übersichtlicheren und saubereren Code sorgen und somit Bugs verhindern. Außerdem können durch Veränderung der festen Werte (im Code, statisch) dadurch bestimmte Spielparameter verändert werden, um das Spiel beispielsweise schwieriger zu gestalten.

Hinweis: Da es sich bei Haskell um eine rein-funktionale Sprache handelt, wurden die Konstanten als normale Werte implementiert, da Haskell eine Änderung von bereits definierten Werten nicht zulässt (keine Seiteneffekte) und somit die Werte konstant bleiben.

```haskell
-- the nearer to 10, the fewer mines
difficulty = 10
```

Der Schwierigkeitsgrad wurde als Konstante definiert. Hierbei kann dieser zwischen 0 und 10 variieren, wobei gilt: je näher der Wert an der 10 liegt, desto weniger Minen lassen sich finden. Dies wurde erreicht, indem innerhalb einer Methode eine zufällige (pseudo-zufällige) Zahl generiert wurde mit der Eigenschaft, dass sie zwischen 0 und 10 liegt. Ist die generierte Zahl größer als die *difficulty* Konstante, dann wird eine Mine erzeugt, sonst eine normale Zelle (mehr dazu in der *genCellPart* Funktion). Es gilt also: Je kleiner der difficulty Wert, deso mehr Minen, desto schwieriger ist das Spiel. Umgekehrt gilt das Gegenteil, da ein difficulty Wert nahe an der 10 nur wenig Minen erzeugt und somit ein leichteres Spiel produziert.

```haskell
-- number of rows and cols
rowNum = 10
colNum = 10
```

Die Konstanten *rowNum* bzw. *colNum* repräsentieren, die Anzahl an Reihen bzw. Spalten die das Spielfeld insgesamt besitzen soll. Diese beiden Werte bestimmen also die Größe des Spielfeldes. Typischerweise sollten die Werte gleich sein, um ein quadratisches Spielfeld zu erzeugen, jedoch sind auch verschiedene Werte möglich. Je größer das Spielfeld ist, desto schwieriger wird es, das Spiel zu gewinnen, da die Wahrscheinlichkeit eine Mine zu erwischen steigt.

```
-- value of mine in cell; mustn't be a value betweed 0-8 as these values are used
for different things
mine = -1
```

Diese Konstante bestimmt lediglich, welcher Integer-Wert die Minen im Spiel repräsentiert, also den Inhalt der Zellen, die als Minen gelten sollen. Dieser Wert darf nicht die Werte 0-8 annehmen, da diese bereits zur Repräsentation von leeren bzw. normalen Feldern dienen.

```
-- debug flag: show all cells
initBool = False
```

Der *initBool* Wert bestimmt mit welcher Variablen die *visibility-flag* der generierten Zellen initialisiert werden soll. Standardmäßig sollte dieser Wert bei *False* liegen, um ein normales Spiel zu ermöglichen. Jedoch kann zu Debugging Zwecken der Wert auch auf True gestellt werden, damit ein Entwickler schnell erkennen kann, wie das Spielfeld aufgebaut wurde. Auch kann so schnellstmöglich das Spiel beendet werden, um zu erkennen wie das Spiel auf bestimmte Aktionen reagiert und Bugs schneller festzustellen.

Funktionen

Der Hauptteil der Implementierung des Spiels in Haskell besteht aus Funktionen. Diese wurden zur Wahrung der Übersichtlichkeit in verschiedene Kategorien unterteilt:

- **Game-Logic Funktionen**: Diese Funktionen beinhalten alle Funktionen, welche sich um die Spiellogik kümmern. Dazu gehören beispielsweise Checks, ob das Spiel gewonnen/verloren wurde oder die Game-Loop, welche in jeder Runde wichtige Aktionen durchführt
- **User-Input Funktionen**: diese kümmern sich um das Einholen von Nutzereingaben z.B. zur Bestimmung der nächsten Aktion
- **Field-Generator Funktionen**: Diese Funktionen kümmern sich um das Generieren des Feldes mit einer zufälligen Anzahl an zufällig platzierten Minen sowie der korrekten Bestimmung der Inhalte der Zellen.
- **Helper Funktionen**: Hilfsfunktionen wie z.B. Getter/Setter. Diese Funktionen kümmern sich um einfache Aufgaben, wie z.B. das zurückgeben vom Zelleninhalt oder String-Konvertierung. Auf diese Funktionen wird in dieser Dokumentation nur wenig eingegangen, da diese größtenteils selbsterklärend sind.

Game-Logic Funktionen

```haskell
-- generates Value
main :: IO ()
main = do
 c' <- (genCells rowNum colNum [])
 cl <- return $ genIntValues c'
 minesweeper (return cl)
```

Die ***Main*-Methode** generiert zunächst das Feld und nutzt dieses dann, um das Spiel zu starten. Dazu später mehr in der *genCells*- bzw. *minesweeper* Funktion.

```haskell
minesweeper :: IO Field -> IO ()
minesweeper field = do
  f <- field
  putStrLn "      0  1  2  3  4  5  6  7  8  9 \n"
  printField f
  -- game over check
  if won f
    then putStrLn "Herzlichen Glückwunsch"
    else
      if lost f
        then putStrLn "Du hast verloren"
        else do
          o <- getOption
          if (o == 1)
            then minesweeper (clickAction f)
            else minesweeper (markAction f)
```

Die ***Minesweeper*-Funktion** ist die Kernfunktion des Spiels. Es nimmt das Spielfeld als Parameter entgegen und „spielt“ darauf. Hier befindet sich die Game Loop, welche in jeder Runde aus mehreren Schritten besteht:

1. Zeichne das Feld
2. Check ob Spieler gewonnen hat
    a. Falls Ja: Methode beenden mit Ausgabe „Herzlichen Glückwunsch“
3. Check ob Spieler verloren hat
    a. Falls Ja: Methode beenden mit Ausgabe „Du hast verloren“
4. Frage Spieler, was er als Nächstes tun möchte (Markieren oder Aufdecken)
5. Wende die richtige Funktion (*markAction*/*clickAction)* auf das Feld an und führe die *Minesweeper* Funktion auf das neue Feld aus. (Rekursion)

```haskell
-- true if game is lost (mine has been clicked on)
lost :: Field -> Bool
lost [] = False
lost (x:xs) = if (isMine x) && (isVisible x)
                then True
                else lost xs
```

Die ***Lost* Funktion** prüft, ob das Spiel bereits verloren wurde. Dazu nimmt es das Spielfeld als Parameter und prüft, ob sich darauf aufgedeckte Minen befinden.

```haskell
-- true if game is won (all mines are marked)
won :: Field -> Bool
won [] = True
won (x:xs) = if isMarked x || (not $ isMine x)
              then won xs
              else False
```

Analog zur Lost Funktion exisitert auch die ***won* Funktion**, die prüft, ob das Spiel bereits gewonnen wurde. Dazu wird geprüft, ob alle Minen markiert wurden.

```haskell
-- mark cell
mark :: Cell -> Cell
mark (i,d,v,True) = (i,d,v,False)
mark (i,d,v,False) = (i,d,v,True)
```

Die ***Mark* Funktion** markiert eine Zelle und gibt sie zurück. Dazu wird die *mark flag* auf True gesetzt.

```haskell
-- click on cell
click :: Cell -> Field -> Field
click _ [] = []
click (i, d, v, m) cl
          | v == True || m == True       = cl
          | otherwise         = if d == 0
                                  -- setze Feldsichtbarkeit True + prüfe alle
adjazenten Zellen -> falls sie auch 0 sind, dann aufdecken
                                  then let step adj cl = case adj of
                                                           [] -> cl
                                                           (x:xs) -> if
(isVisible x) -- sichtbare Zellen ignorieren

then step xs cl
```

```
else

if (not ((getValue x) == 0)) -- Werte != 0 -> aufdecken

then step xs (editCell x (setVisible x) cl)

else step xs (click x cl)  -- Werte == 0 -> aufdecken + adjazente Zellen aufdecken
                                          in step (getAdjacentCells (i,d,v, m)
cl) (editCell (i,d,v, m) (i,d,True, m) cl)
                                       else (editCell (i,d,v, m) (i,d,True,m) cl)
```

Die ***Click* Funktion** dient dem Aufdecken einer Zelle. Sie ist etwas komplexer, da Minesweeper das Feature besitzt, dass, bei Aufdeckung einer Zelle, alle adjazenten Zellen ebenfalls aufgedeckt werden, die nicht adjazent zu einer Mine liegen. Dies geschieht Rekursiv, da die neu aufgedeckten Zellen ebenfalls ihre Nachbarzellen aufdecken, außer sie befinden sich neben einer Mine.

User-Input Funktion

```
getOption :: IO Integer
getOption = do
  putStrLn "Bitte gib deine nächste Aktion ein: \n(1) - Zelle aufdecken\n(2) -
Zelle (de)markieren"
  o <- getLine
  return $ read o
```

Die ***getOption*-Funktion** dient dazu, die nächste Spieleraktionen aufzunehmen. Es wird entsprechend markiert bzw. aufgedeckt.

```
getIndex :: IO Index
getIndex = do
  putStrLn "Gebe die Nummer der Reihe an"
  r <- getLine
  putStrLn "Gebe die Nummer der Spalte an"
  c <- getLine
  return ((read r), (read c))
```

Die ***getIndex* Funktion** holt sich zwei Eingaben vom Nutzer, bei welchen es sich um die Koordinaten einer Zelle handelt und gibt diese als Wert vom Datentypen *Index* zurück.

Field Generator Funktionen

```
-- generate cells
genCells :: Integer -> Integer -> [Cell] -> IO [Cell]
```

```
genCells w h l = do
  c <- genCellPart (w',h)
  if w < 1
    then return l
    else
      if h < 1
        then genCells (w-1) 9 (c:l)
        else genCells w (h-1) (c:l)
  where
    w' = w - 1
```

Die ***genCells*-Funktion** generiert die Zellenrümpfe und erstellt so ein vorläufiges Feld. Parameter sind die Höhe und die Breite des Feldes, sowie eine Liste an Zellen, wobei die Zellenliste zu Anfang leer sein darf und dann von der Methode gefüllt wird. Mittels der *genCellPart* Methode generiert sie Zellenrümpfe, welche entweder eine Mine sind (nach Zufallsprinzip entschieden über *difficulty* Konstante (siehe S. 3)) oder eine Zelle ohne Inhalt. Die leeren Zellen werden im Nachhinein durch die *genIntValues* gefüllt, also jede Zelle, die keine Mine ist, bekommt als Inhalt die Anzahl der adjazenten Minen zugewiesen.

Helper Funktionen

Da es sich bei den Helper Funktion um relativ kurze und einfache Methoden handelt, wird im Rahmen dieser Dokumentation nur auf die wichtigsten Methoden eingegangen.

```
get :: Index -> Field -> Cell
get (i,j) [] = ((-1,-1), 0, False, False)
get i (x:xs)
        | (index' x) == i     = x
        | otherwise               = get i xs
```

Die ***get* Funktion** bekommt als Parameter einen Index und ein Feld zugewiesen und gibt dann die Zelle mit den, in Index enthaltenen Koordinaten zurück.

```
getRandNum :: Integer -> IO Integer
getRandNum max = do
    num <- randomRIO (0, max)
    return num
```

Die *getRandNum* Funktion generiert eine zufällige Zahl zwischen 0 und max. Diese Methode wird verwendet, um (pseudo-)zufällige Entscheidungen zu fällen, die eine gewisse Wahrscheinlichkeit besitzen.