Universität Bremen

Praktische Informatik 3 WS 2023/24

Thomas Barkowsky
Tede von Knorre
Yaroslav Purgin
Tim Sperling

8. Übungsblatt

Ausgabe: 18.12.2023
Abgabe: 15.01.2024

Du entscheidest, dass du dich als aufstrebender Haskell-Programmierer in der Spieleindustrie ausprobieren möchtest. Für dein erstes Projekt willst du ein kleines rundenbasiertes Spiel im Terminal erstellen und nimmst als Inspiration ein Spiel, das du damals im erstem Semester deines Studiums programmiert hast.

Ein Spielfeld wird aus Kacheln bestehen, die wie folgt definiert sind:

```
data Direction = West | East | North | South deriving (Show, Eq)
data Tile = Road | Grass | Goal | Player Direction | Enemy Direction deriving Eq
```

`Player` ist die gesteuerte Spielfigur, die in eine der vier `Direction`'s gedreht sein kann. `Enemy` ist der Gegner, der seine Strecke hin und zurück ablaufen wird. `Road` ist, wo der Spieler und die Gegner sich frei bewegen können. Auf `Grass` kann man nicht laufen, dies sind also leere Kacheln.

```
type Position = (Int,Int)
newtype Field = Field { tiles :: (M.Map Position Tile) } deriving Eq
```

Das Spielfeld ist eine Abbildung von Positionen auf einzelne Kacheln. Positionen sind dabei Paare (x, y).

```
data GameState = Playing | NoMove | Won | Lost deriving (Show, Eq)
```

`GameState` sagt etwas darüber aus, in welchem Zustand das Spiel momentan ist. Mehr dazu später.

Für Windows-Nutzer kann es in dieser Übung Probleme bei der Darstellung geben. Deswegen wurde `puzzle_game.bat` angelegt, die das Terminal öffnet, die Codepage zu Unicode wechselt und erst dann `stack ghci` startet.

8.1 *Ein Spielfeld aufbauen* — *5 Punkte*

Wir fangen mit den Funktionen an, die für uns das Spielfeld aufbauen. `createField` erzeugt ein mit leeren Kacheln gefülltes Spielfeld. Hierbei geben die beiden Parameter die Breite und Höhe des Spielfeldes an. Die generierten Koordinaten starten bei 0 und hören bei n-1 auf. Beachtet, dass `Show` für `Tile` und `Field` erst später implementiert werden. In der Zwischenzeit könnt ihr selbst die Aufzählungstypen um `Show` im **deriving** ergänzen.

```
createField :: Int → Int → Field
createField 3 2 ⇝ Field {tiles = fromList [((0,0),Grass),((0,1),Grass),((1,0),Grass),
                                           ((1,1),Grass),((2,0),Grass),((2,1),Grass)]}
```

`placeRoad` baut für uns die Wege. Zwei Positionen werden erwartet, Start- und Endpunkt. Es reicht, wenn die Funktion nur gerade Wege bauen kann. Mit (3,3) und (3,12) wird der vertikaler Weg von y=3 bis y=12 erzeugt und mit (3,3) und (15,12) ein horizontaler von x=3 bis x=15.

```
placeRoad :: Field → Position → Position → Field
```

`placePlayer` und `placeEnemy` positionieren die Spielfigur und die Gegner mit den gegebenen Positionen und Richtungen. Beide können nur auf einem Weg platziert werden, sonst wird ein unverändertes Feld zurückgegeben.

```
placePlayer :: Field → Position → Direction → Field
placeEnemy :: Field → Position → Direction → Field
```

Mit `placeGoal` wird der Zielpunkt gesetzt. Hier gilt auch, dass es nur auf einem Weg platziert werden kann.

```
placeGoal :: Field → Position → Field
```

Hinweise:

- Um das Board erfolgreich zu manipulieren, setzt euch mit der Dokumentation für Data.Map auseinander. Hilfreich sind besonders die Funktionen empty, lookup, !, insert, adjust, fromList.
- Es kann sinnvoll sein die Aufgabe 1 gleichzeitig und abwechselnd mit der Aufgabe 2 zu implementieren, da man den Zustand des Spielfelds schlecht ohne eine visuelle Darstellung deuten kann.

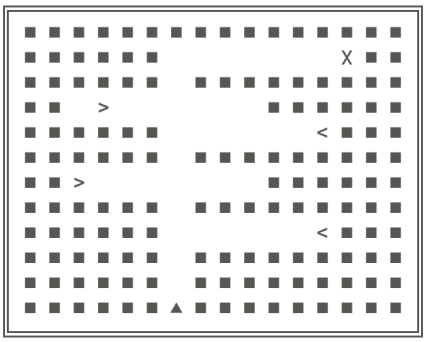

Abbildung 1: Das Spielfeld als Ausgabe von `displayField`

8.2 *Wie sieht's denn eigentlich aus?* *4 Punkte*

Der erste Schritt, um das Spielfeld darstellen zu können, besteht darin, die Darstellung für einzelne Kacheln zu implementieren.

Die Zeichen, die ihr für die Darstellung benutzen könnt, sind in `symbols` vorgegeben.

```
instance Show Tile where
show...

putStrLn (show (Enemy East)) ⇝ ">"
```

Dann kann die Darstellung für das ganze Feld gemacht werden. Show würde das Spielfeld mit Leerzeichen zwischen jeder Kachel darstellen, allerdings ohne dem Außenrand wie in der Abbildung 1.

```
instance Show Field where
show...
```

`fieldFromString` und `fieldToString` können ein Spielfeld in einen String umwandeln und umgekehrt. Diese Funktionen werden später fürs Lesen und Schreiben aus den/ in die Dateien benutzt. `fieldToString` ist ähnlich zu `show`, aber erzeugt keine Leerzeichen zwischen den Kacheln, um wertvollen Speicherplatz zu sparen.

```
fieldFromString :: String → Field
fieldToString :: Field → String
```

`displayField` soll die Dartsellung wie in der Abbildung 1 erzeugen.

```
displayField :: Field → String
```

8.3 *In Bewegung setzen!* *4 Punkte*

Nun möchten wir etwas Dynamik. Die Steuerung für die Spielfigur wird in `movePlayer` implementiert. Die Funktion gibt ein Tupel mit dem neuen Feld zurück und einen der vier möglichen `GameState`'s:

1. `Playing`, falls ein Schritt gemacht werden konnte und der Spieler auf `Road` steht.
2. `Lost`, falls der Spieler beim Zug auf eine Kachel mit einem Gegner gelandet ist.
3. `Won`, falls der Spieler das Ziel erreicht hat.
4. `NoMove`, wenn es einen Versuch auf `Grass` zu gehen gab. Die Spielfigur soll sich in diese Richtung drehen, aber sich nicht von der Kachel wegbewegen. Für die Gegner gilt dann auch: kein Zug.

```
movePlayer :: Field → Direction → (Field, GameState)
```

Die KI des Gegners folgt ähnlichen Regeln wie die des Spielers. Die Gegner können nicht auf Grass laufen. Außerdem dürfen sie nicht auf die Kachel mit dem Ziel oder auf Kacheln mit anderen Gegnern. Wenn ein Gegner keinen Schritt vorwärts machen kann, benutzt er den Zug, um sich um 180° zu drehen. Da es auf der Karte mehrere Gegner geben kann, soll direkt die Funktion implementiert werden, die mit einem Aufruf alle Gegner bewegt. (Eine Hilfsfunktion ist aber empfohlen.)

Folgende `GameState`'s sind möglich:

1. `Playing`, falls keiner von den Gegnern auf den Spieler gestoßen ist.
2. `Lost`, falls einer der Gegner mit der Kachel des Spielers kollidiert ist.

```
moveEnemies :: Field → (Field, GameState)
```

8.4 *Kommunikation mit der Außenwelt* *7 Punkte*

Im Modul `Main` werden Ein- und Ausgabe behandelt. Mit `save` und `load` können Spielfelder gespeichert und auch nach einem Neustart des Programms geladen werden. Es wird im Ordner `fields` mit dem Dateiformat `.field` gespeichert. Als Eingabe reicht es, nur den Namen der Datei ohne die Endung anzugeben.

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

Um einen String in eine Datei zu schreiben bzw. von dort zu lesen, kann man folgende vordefinierten Funktionen nutzen. Denkt daran, Fehler abzufangen!

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

Das Spiel soll beim Starten ein Spielfeld wie in der Abbildung 1 anzeigen und auf die Eingabe vom Nutzer warten. `System.Console.Haskeline` gibt uns die Funktionen vor, die ein Zeichen als Eingabe auslesen können ohne auf die Betätigung der Enter-Taste zu warten. Mit den W-A-S-D-Tasten kann man die Eingabe tätigen. W würde den Spieler in die Richtung Norden bewegen. Es reicht, wenn nur kleingeschriebene Buchstaben als Eingabe erwartet werden. Nachdem die Eingabe erfasst und der Zug ausgeführt wurde, werden auch die Gegner bewegt. Das Spiel endet, wenn der Spieler entweder das Ziel erreicht oder mit einem Gegner kollidiert. Eine Nachricht soll angezeigt werden, die erklärt mit welchem Zustand das Spiel beendet wurde. Danach soll ein neues zufälliges Spielfeld ausgewählt und geladen werden. Es sorgt für eine angenehmere Darstellung, wenn ein bisschen Zeit zwischen den Zügen und nach dem Spielende gelassen wird.

Zu jedem Zeitpunkt soll es möglich sein, ein Options-Menü mit der Taste Q zu öffnen. Dort kann man mit den Tasten 1-4 aus folgenden Optionen wählen:

```
#Options:
1. Continue
2. Save
3. Load
4. Quit
```

`Continue` schließt das Options-Menü wieder. `Save` fragt nach dem Dateinamen und speichert dann das Spielfeld ab. `Load` fragt auch nach dem Dateinamen und lädt das angegebene Spielfeld; wenn es fehlschlägt, wird `defaultField` geladen. Mit `Quit` verlässt man das Spiel.

`main` ist der Startpunkt des Programms, wo `play` aufgerufen wird. `play` erfasst die Eingaben des Nutzers, evaluiert diese und ruft sich selbst am Ende rekursiv auf.

```
main :: IO ()
play :: Field → IO ()
```

Hinweise:

- Vorgegeben sind `getInputT` und `getLineInputT` die wie folgt abgefragt werden können: `input <- runInputT defaultSettings getInputT`.
- `listDirectory` aus `System.Directory` listet alle Dateien im Ordner auf.
- `randomRIO(0, 4)` aus `System.Random` liefert eine zufällige Zahl zwischen 0 und 4.
- `threadDelay` aus `Control.Concurrent` kann das Programm zum Warten zwingen.

8.5 *Bonus* *1-5 Punkte*

- **1 Punkt:** Implementiere zusätzlich ein Startmenü, das die Funktionalität des Pause-Screens hat und zusätzlich Spielfeld-Auswahl ermöglicht.
- **3-4 Punkte:** Überlege dir und implementiere eine Spielfunktionalität, die das Spiel **sinnvoll** ergänzt. Zum Beispiel:
  - Teleporter oder
  - Sammelobjekte, ohne welche man das Level nicht abschließen kann.