

## Hausaufgabe Haskell (30 Punkte)



Die Hausaufgabe befasst sich mit der Entwicklung eines Bots für das Spiel Ploy (mit leichten Abwandlungen) in Haskell. Zur Implementierung wird die Vorlage eines Stack-Projekts zur Verfügung gestellt, das bereits eine Grundstruktur vorgibt sowie Funktionen als Schnittstelle zur Bewertung beinhaltet. Diese Schnittstellen dürfen nicht verändert werden. Zusätzlich zur Programmierung in Haskell beinhaltet die Hausaufgabe Anteile der Qualitätssicherung (Testen und Metriken).





Viel Erfolg!

### Ploy

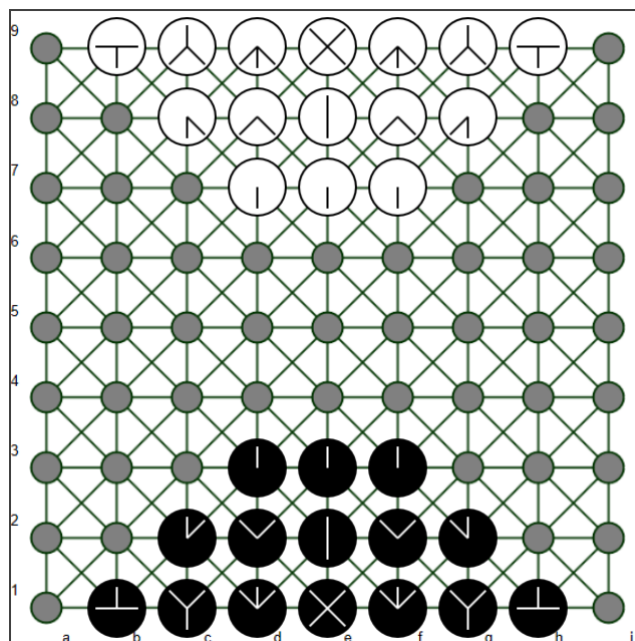
Ploy ist ein Brettspiel ähnlich Schach, in dem jede\*r Spieler\*in eine Menge Figuren mit verschiedenem Bewegungsfreiraum besitzt, mit denen gegnerische Figuren geschlagen werden können. Ziel ist es, entweder den *Commander* oder alle anderen Figuren der gegnerischen Partei zu schlagen. Ploy gibt es für zwei oder vier Spieler\*innen, wobei nur die Variante für zwei Spieler\*innen Teil der Aufgabe ist.

### Regeln

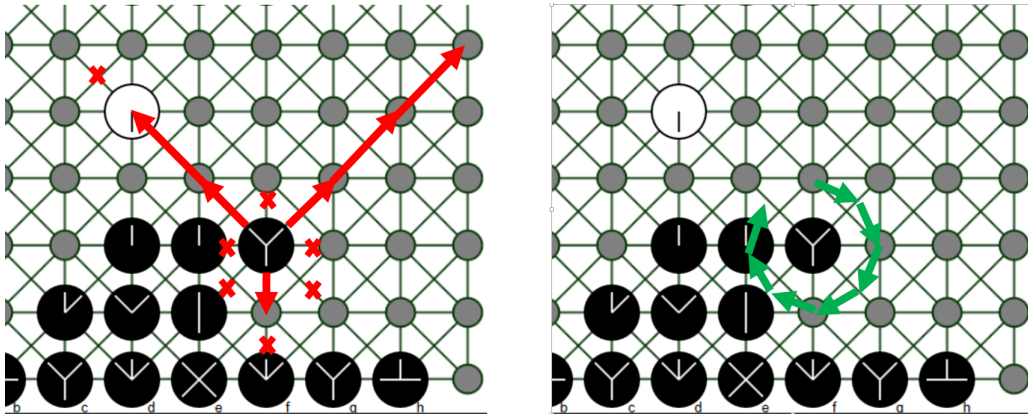
Gespielt wird auf einem zweidimensionalen Spielbrett mit 9x9 Feldern - schwarz beginnt. Jede Partei besitzt am Anfang 15 Figuren, auf denen unterschiedlich viele Bewegungsrichtungen durch Striche markiert sind. In jedem Zug kann eine Figur bewegt und/oder gedreht werden, wobei sich die Fähigkeiten der einzelnen Figuren unterscheiden. Konkret gibt es die folgenden Figuren:

Figur	Varianten	Distanz Bewegungen / Drehung
Shield		$\leq 1$ Feld <b>und/oder</b> Drehung ( <b>danach</b> )
Probe		<b>entweder</b> bis zu 2 Felder <b>oder</b> Drehung
Lance		<b>entweder</b> bis zu 3 Felder <b>oder</b> Drehung
Commander		<b>entweder</b> $\leq 1$ Feld <b>oder</b> Drehung

Das Startfeld sieht dabei wie folgt aus:



In jedem Zug muss eine Bewegung und/oder eine Drehung durchgeführt werden, mit dem eine Veränderung des Zustands herbeigeführt wird. Die Bewegung einer Figur ist horizontal, vertikal und diagonal entlang der auf der Figur abgebildeten Richtungslinien möglich. Dabei können Figuren nicht übersprungen werden. Im Zielfeld darf entweder keine Figur oder eine gegnerische Figur sein, wobei im zweiten Fall die gegnerische Figur vom Spielfeld entfernt wird. Die folgende Grafik zeigt die Zugmöglichkeiten der Y-Lance:



Das Ende des Spiels ist erreicht, wenn eine der beiden Parteien entweder keinen Commander mehr besitzt, oder nur noch den Commander. Die Partei, die den letzten Spielzug durchgeführt hat, gewinnt.

## Notation

Für die Beschreibung der Spielsituation und Spielzüge verwenden wir eine abgewandelte FEN-Notation (bekannt von Schach). Das heißt, die Spalten werden mit kleinen Buchstaben von a bis i beschriftet, die Reihen mit Zahlen von 1 bis 9. Um ein Feld eindeutig zu bestimmen, wird immer zuerst die Spalte und dann die Zeile angegeben. Dabei bezeichnet a1 die untere linke Ecke, i9 die obere rechte. Zu Beginn stehen die schwarzen Figuren auf den “unteren” Feldern in den Reihen 1-3, die weißen auf den “oberen” in Reihe 7-9.

## Spielbrett

Das Spielbrett wird durch einen String beschrieben, der wie folgt aufgebaut ist: Reihen werden mit “/” getrennt, und innerhalb der Reihen werden die Felder mit “,” getrennt. Am Anfang und am Ende des Strings gibt es kein “/”, und am Anfang und Ende jeder Reihe auch kein “,”. Zuerst wird das Feld a9 angegeben, zuletzt das Feld i1. Innerhalb eines Feldes kann eine Figur stehen, die wie folgt codiert ist: Das erste Zeichen ist die Farbe, “b” steht für schwarz und “w” steht für weiß. Die nachfolgenden Stellen sind eine Dezimalzahl zwischen 1 und 255. Diese codiert, als vorzeichenlose 8-bit Binärzahl aufgefasst, mit jedem Bit eine Richtung, in die Figur gehen darf (in den Grafiken als Strich dargestellt). Das LSB (Least Significant Bit) steht dabei für oben (bzw. Norden, oder entlang einer Spalte in Richtung der Reihe mit der höheren Zahl), und die jeweils höheren Bits stehen für die anderen Richtungen, im Uhrzeigersinn von Norden ausgehen. Beispiel: b21, also 0b00010101, beschreibt die folgende Figur:



Wenn keine Figur auf dem Feld steht, bleibt auch die Stelle im String leer.

Das Spielbrett zu Beginn wird also wie folgt als gültiger FEN-String beschrieben:

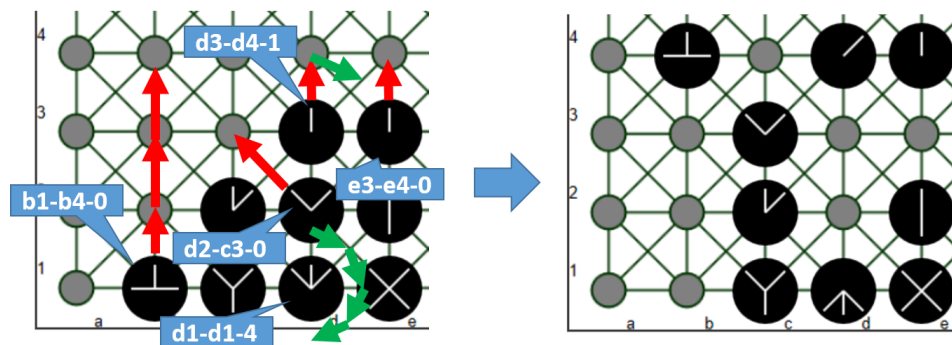
```

,r84,w41,w56,w170,w56,w41,w84,/,,w24,w40,w17,w40,w48,,/
,,,w16,w16,w16,,,/,/,/,/,/,/,/,/,/,/,b1,b1,b1,,,/
,,,b3,b130,b17,b130,b129,,,/b69,b146,b131,b170,b131,b146,b69,

```

## Züge

Züge werden durch einen String der Form `<start>-<ziel>-<drehung>` beschrieben, wobei `<drehung>` die Anzahl der Dreh-Schritte im Uhrzeigersinn beschreibt. Eine Drehung, die bewirkt, dass eine Linie statt nach Norden nun nach Süd-Osten zeigt, würde durch `<drehung>=3` beschrieben. Eine Drehung um 1 gegen den Uhrzeigersinn entspricht in unserer Notation der `<drehung>=7` im Uhrzeigersinn. Drehungen um mehr als sieben Schritte sind nicht erlaubt. Entsprechend den bestehenden Möglichkeiten hat ein Zug also immer sieben Zeichen. Ein paar Beispiele:



## Aufgabestellung

Das Ziel der folgenden Aufgabenstellung ist die Entwicklung eines Bots für Ploy. Zur Implementierung ist eine Vorlage in Form eines Haskell Stack-Projekts verfügbar. Dieses muss als Grundstruktur zur Implementierung verwendet werden. Zur Implementierung des Ploy-Bots werden 7 Schnittstellen in Form von Funktionssignaturen vorgegeben. Jede Funktion wird anhand ihrer funktionalen Anforderungen bewertet (*FP: funktionale Punkte*), die unten genauer beschrieben werden. Zusätzlich gibt es einen nicht-funktionalen Anteil, indem die Implementierung des Bots getestet werden soll. Der von euch mit Unit-Tests geprüfte Anteil der Implementierung wird anhand der *Haskell Program Coverage (HPC)* ermittelt.

Implementiert und testet die folgenden Schnittstellen im *src* und *test* Ordner:

1. Modul Board: `validateFEN :: String -> Bool` 2 FP + 1 CP
- Die Funktion erhält einen String und überprüft diesen anhand des beschriebenen FEN-Formats für Ploy (s. Abschnitt Spielbrett). Dabei gibt die Funktion genau dann `True` zurück, wenn es sich um einen gültigen FEN-String handelt. Die Gültigkeit der Spielsituation muss nicht überprüft werden.

2. Modul Board: `buildBoard :: String -> Board` 2 FP + 1 CP  
 Die Funktion erhält einen geprüften und somit gültigen FEN-String, aus dem der Spielzustand des vorgegebenen Typs `type Board = [[Cell]]` erzeugt werden soll. Dabei entspricht das Element des Zustands `(board!!0)!!0` der Ecke oben links (Feld a9). Das Element `(board!!0)!!8` entspricht der Ecke oben rechts (Feld i9). Hinweis: der `!!`-Operator greift auf den Index einer Liste zu.
3. Modul Board: `line :: Pos -> Pos -> [Pos]` 3 FP + 1 CP  
 Die Funktion erhält zwei Argumente (1: Startposition, 2: Zielposition). Zurückgegeben wird eine Liste von Positionen, die die Linie vom Start- zum Zielfeld darstellt. Positionen des vorgegebenen Typs `Pos` entsprechen den Bezeichnern des Spielbretts (a1 bis i9). Der Rückgabewert ist sortiert und beinhaltet als erstes Element die Startposition und als letztes Element die Zielposition. Es kann davon ausgegangen werden, dass übergebene Positionen innerhalb des Feldes liegen und Start- und Zielposition auf einer horizontalen, vertikalen oder diagonalen Geraden liegen.
4. Modul Ploy: `gameFinished :: Board -> Bool` 3 FP + 1 CP  
 Die Funktion überprüft, ob das Spiel im übergebenen Zustand beendet ist. Der Zustand (`Board`) entspricht dabei einer gültigen Spielsituation. Die Funktion gibt genau dann `True` zurück, wenn kein weiterer Zug erlaubt ist.
5. Modul Ploy: `isValidMove :: Board -> Move -> Bool` 5 FP + 1 CP  
 Die Funktion erhält einen Spielzustand (`Board`) und einen Zug des vorgegebenen Typs `Move`. Der Rückgabewert ist genau dann `True`, wenn der Zug im aktuellen Zustand durchgeführt werden kann. Es kann davon ausgegangen werden, dass ein gültiger Zustand übergeben wird. Der Zug enthält eine mögliche Bewegung (Drehung und Schritte) der Figur vom Startfeld des Zustands.
6. Modul Ploy: `possibleMoves :: Pos -> Cell -> [Move]` 6 FP + 1 CP  
 Die Funktion ermittelt für das übergebene Feld (`Cell`) auf der übergebenen Position (`Pos`) alle möglichen Spielzüge und gibt diese als Liste zurück. Dabei wird kein expliziter Spielzustand berücksichtigt. Die Parameter `Pos` und `Cell` beinhalten gültige Werte. Züge müssen eine Änderung des Zustands bewirken und dürfen nicht mehrfach vorkommen. Züge mit unterschiedlicher Rotation bei gleicher Zustandsänderung (bspw. Commander dreht um 1 bzw. 3 Schritte) zählen als verschiedene Züge.
7. Modul Ploy: `listMoves :: Board -> Player -> [Move]` 2 FP + 1 CP  
 Die Funktion ermittelt für einen Spieler in einem zulässigen Spielzustand alle durchführbaren Züge. Wie bereits bei `possibleMoves` gilt: Züge müssen eine Änderung des Zustands bewirken und dürfen nicht mehrfach vorkommen. Züge mit unterschiedlicher Rotation bei gleicher Zustandsänderung zählen als verschiedene Züge.

## Hinweise zur Bearbeitung

**Implementierung:** Wir stellen euch einfache Validierungstests zur Verfügung. Diese können mit `stack test ploy:validate` ausgeführt werden. Die Hausaufgabe wird nur bewertet, wenn die Tests mit dem Befehl ausgeführt werden können. Schnittstellen werden nur bewertet, wenn die zugehörigen Validierungstests erfolgreich durchlaufen. Die Validierungstests **dürfen nicht geändert** werden.

Die Implementierung der funktionalen Anforderungen erfolgt ausschließlich in den vorgegebenen Modulen *Board* und *Ploy* im Ordner *src*. Dabei darf der bereitgestellte Code nicht verändert werden. Ausnahmen sind: Modulimporte, die Implementierung der Schnittstellen und Instanziierungen der Typklasse *Show* anstatt den vorgegebenen *deriving Show*. Es dürfen nur Module und deren Funktionen aus den Paketen *base* und *split* importiert werden. Die Stack-Konfigurationsdateien dürfen nicht verändert werden. Netzwerkzugriffe und die Einbettung von anderen Programmiersprachen sind nicht erlaubt.

Funktionen zur Verarbeitung und Analyse von Bits befinden sich im Modul `Data.Bits`. Nützlich sind hier die Funktionen `popCount` und `testBit`. Die vorgegebene Funktion `rotate` im *Ploy*-Modul berechnet die Drehung einer Spielfigur `o` um `tr`-Schritte.

Damit ihr eure Implementierung testen könnt, stellen wir euch einen Webserver zum Download auf ISIS zur Verfügung. Hinweise zur Verwendung findet ihr in der Readme. Mithilfe von `stack exec ploy` erzeugt ihr eine ausführbare Datei eures Bots, bei dem in der `main`-Methode im Ordner *app* ein zufälliger Zug aus der Liste aller Züge ausgewählt wird. Die Züge werden durch eure `listMoves`-Funktion berechnet.

**Testen und Haskell Program Coverage:** Eure Unit-Tests implementiert ihr im Ordner *test* mithilfe von `Test.HSpec` (wie in den Übungen und den Validierungstests). Damit ihr *Coverage Punkte* durch eure Tests erhalten könnt, müssen eure Tests mithilfe von `stack test ploy:units` ausführbar sein. Coverage Punkte werden anteilig zur gesamten erreichten Haskell Program Coverage ( $= \sum \text{covered} / \sum \text{total}$ ) der Module `Board` und `Ploy` vergeben. Die anteiligen *Coverage Punkte (CP)* der Schnittstelle erhält man dabei nur, wenn mindestens die Hälfte der zugehörigen funktionalen Punkte erreicht wurden. Die Abdeckung eurer Implementierung durch eure Tests können durch den Befehl `stack test --coverage ploy:units` berechnet werden.

**Abgabe des Projekts:** Zur Abgabe eures Projekts erzeugt ihr ein übliches zip-Archiv mit dem Dateinamen "`{vorname}-{nachname}-{matrikelnummer}-ploy.zip`". Das Archiv beinhaltet als einziges den Ordner *ploy* des Stack-Projekts. Der Ordner beinhaltet genau die Dateien der Vorgabe. Zusätzlich dürfen nur *.hs*-Dateien im Ordner *test* enthalten sein (falls wirklich benötigt). Zuletzt ladet ihr die zip-Datei auf ISIS hoch und seid fertig (mehrfach Upload erlaubt). Verspätete Abgaben werden mit 0 Punkten bewertet.