Search Topics

Overview

Bookmarks

Course Schedule

Table of Contents 5

Course Information

Course Schedule 4

Haskell Group Assignment

Java Group Assignment: Reflective Parser

Tutorial Materials

In-lecture tasks 1

Haskell Group Assignment

Print

The object of this assignment is to write a program that plays the board game, Apocalypse. Now, if we were game developers that would be one thing, and it would be clear what we need to do. However, we are *not* game developers, we're computer scientists, so we don't want to actually play the game *ourselves*: we might as well let the *computer* play the game on our behalf! So we are going to write the game such that we can pick a game-playing strategy from the a selection of strategies for both the black and white players and let the strategies play the game for us. Much less effort! Right? Of course, we might actually want to play too, so that will be an option too, but we'll just call this a strategy called "human".

Warning: This specification is subject to update.

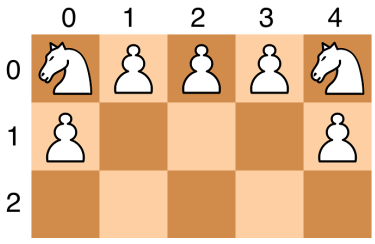
- Learning Objectives
- What is Othello?
- Functional Requirements
- Non-functional Requirements
- Notes
- Handing in this Assignment

Learning Objectives

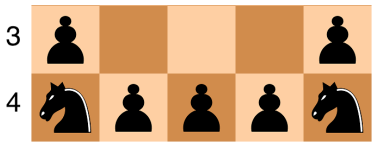
1. Learn to learn a new language on your own.
2. Learn simple IO concepts in Haskell.
3. Learn to manipulate Lists is Haskell.
4. Learn patterns in Haskell.
5. Learn how to write non-trivial functions in Haskell.
6. Learn how to handle "state" information (by passing it around) in Haskell.

What is Apocalypse?

As already mentioned, Apocalypse is a board game, a variant of chess, played on an 5x5 grid with only knights and pawns. For details of the game, see the Wikipedia page



on Apocalypse.



- **Initial board state:** The initial board state is as pictured in the figure at right.
- **Moving pieces**. Moves by the knights and pawns are the same as for chess, except that spawns do not have the double-step initial move.
- **Turn-taking**. The game moves take place in rounds rather than turns, that is, each player chooses a move without knowing the other's move, and they both move at the same time.
- **Clashes**. If two pieced both move to the same square in a round, a knight captures a pawn. Same-type pieces are *both* removed from the board.
- **Pawn missed capture**. If a capture was declared using a pawn, but the piece to be captured moved from its square, the pawn move still stands. (The move converts to a diagonal step instead of a capture.)
- **Penalties**. If a declared move is illegal, the player incurs a penalty point.
- **Pawn promotion**. A pawn promotes to knight when reaching the last rank, but only when the player has less than two knights. Otherwise the player must redeploy the pawn to any vacant square.

The game ends when an of the following conditions arrises:

- One of the players looses all his/her pawns. The other player is the winner.
- One of the players accumulates two penalty points. The other player is the winner.
- Both players pass on the same round. The one with the most pawns wins.

The last rule above is added because we are playing automated strategy-against-strategy, and also want to avail ourselves to easy testing, so we need an "abort" method.

Functional Requirements

1. **Program Name.** The file name for the main program will be "Apoc.hs", the program may include module files in the same directory, which must submitted with the main program.
2. **Run modes**. The program can be run either as a command line program or from the interactive environment by typing "`main`".
3. **Command line**. If the program is run from the command line, it can take either 0 or 2 arguments. If there no arguments, then the programs goes into interactive mode

(see the "interactive mode" requirement). If there are two arguments, then these are to be taken as a strategy name for the black and white players respectively. If these are both legal strategy names, then the program continues by playing the two strategies against one another. If one or both of these are not legal strategy names, then the program prints a list of legal strategy names (see requirement "printing strategy names") and quits.

4. **Interactive startup**. If the program is run from within the interactive environment by typing "`main`", then the program will immediately enter interactive mode (see the "interactive mode" requirement).
5. **Interactive mode**. If the program enters interactive mode then it will print a descriptive message, including the list of possible strategy names (see requirement "printing strategy names"). It will then prompt for a black strategy name, and then prompt for a white strategy name. If one are both of these are not legal strategy names, then the program prints a list of legal strategy names and quits. If these are both legal strategy names, then the program continues by playing the two strategies against one another.
6. **Printing strategy names**. Whenever the program prints a list of strategy names, it will print each strategy on a separate line with exactly two spaces (only) before the name of each strategy. A newline shall immediately follow each strategy name. No other program output shall begin a line with exactly two spaces followed immediately by a non-space character.
7. **One game**. If the program obtains the two legal strategy names, it will play exactly one game and then terminate.
8. **Game play**. The game is played by playing each round with a play by both the black and white strategies (a function `type Chooser = GameState -> PlayType -> Player -> IO (Maybe [(Int,Int)])`.) by a call of the form `(chooser state Normal player)`. If one or both of the players moves a pawn into the far row from which they started, then a second (sorta) round is played in which the pawn(s) is (are) either upgraded to a knight (automatically without calling the strategy, but showing the move in the output) or the pawn is placed by a call of the form `(chooser state PawnPlacement player)` which returns a singleton list contain the coordinate (x,y) pair indicating the chosen coordinate of the pawn.
9. **Coordinates**. The coordinate system we shall use to describe the cells of the board range from 0 to 4 where the leftmost column is labeled 0, and the topmost row is labeled 0. The column number is always listed first, and a general cell location is often written (x,y) where x and y are integers between 0 and 4 inclusive. This system is dedicated in the board figure above.
10. **Game tracing**. The program must print out a trace of the game starting with the initial state (the board for which you

can get from calling ApocTools.initBoard). The format of the state must include the state of the board in the form:

```
>>>
black play-description
white play-description
 _ _ _ _ _
|X|/|/|/|X|
|/|_|_|_|/|
|_|_|_|_|_|
|+|_|_|_|+|
|#|+|+|+|#|
```

That is, three greater-than signs followed by a new line, the two play-description lines separated by a newline and followed by a newline, followed immediately by an intro line consisting of <space><underscore> repeated 5 times, and then each of 5 lines consisting of alternating vertical bars and cell characters ("X" for a white knight, "/" for a white pawn, "#" for a black knight, "+" for a black pawn, and an underscore for a empty space, all separated, starting and ending with a vertical bar. There can be no characters (white space or otherwise) before or after lines these lines and these lines must be consecutive (no lines interspersed between them). The <play-description> line is of the form:

(*played*, *penalty*)

with no space before or after the parentheses, where *penalty* may be 0, 1, or 2, and *played* may be one of

- - "Played ((*fromX*,*fromY*),(*toX*,*toY*))" (where *fromX*. fromY, *toX*, and *toY* are integers in the range 0..4 inclusive): Indicating a legal move from (*fromX*,*fromY*) *to* (*toX*,*toY*)
- - "Init": Indicating the initial state of the board. This should occur only in the first state output.
- - "Passed": Indicating that the player passed on its move.
- - "Goofed ((*fromX*,*fromY*),(*toX*,*toY*))": Indicating the player tried to perform an illegal move on a "normal" turn.
- - "UpgradedPawn2Knight (X,Y)": Indicating that a pawn was upgraded to a knight.
- - "PlacedPawn ((*fromX*,*fromY*),(*toX*,*toY*))": Indicating that a pawn was legally placed as a result of being upgraded.
- - "BadPlacedPawn ((*fromX*,*fromY*),(*toX*,*toY*))": Indicating the player tried to do an illegal pawn placement as a result of an upgrade.
- - "NullPlacedPawn": Indicating that the player returned a pass in response to pawn upgrade move.
- - "None": Indicating that this player was playing in

opposition to the other player doing a pawn upgrade (pawn-to-knight, or pawn placement).

No other lines in the output shall begin with the characters "(", " " (space), or ">", including any "conversational" lines in the interactive mode of the program (see the interactive mode requirement).

Note: The `show` function associated with the `GameState` type in Apoc`Tools` will easily achieve this for you -- it has implemented this for you already. For an example, see my transcript of a greedy strategy vs itself, and of a human strategy agains a greedy strategy (this later file depicts the human strategy as specified in the human requirement below).

It is absolutely critical that you follow this requirement exactly as we will use a program that parses your output to check your work.

11. **Strategies**: You should implement at least 2 different strategies to choose from. All strategies should have a type of `type Chooser = GameState -> PlayType -> Player -> IO (Maybe [(Int,Int)])`. Where returning a value of `Nothing` indicates a pass.
12. **"human" strategy**: Besides the two automated strategies, you must implement a strategy called "human" which isn't really a strategy at all, but asks for input from standard input (stdin) to determine the moves. This strategy must conform to the following requirements:
    a. **Normal move**. For normal moves, the input from the user must be in the form <Int1><sp><Int2><sp><Int3><sp><Int4> (all on one line). e.g.: the line "1 3 1 2" is a valid black move if a black pawn is at location (1,3) and (1,2) is clear. However, "1 3 1 1" is never a valid move as it does not describe a legal move, and "1 5 1 4" is not valid because the 4 is out of range.
    b. **Pawn placement move**. For pawn placement moves the input must be in the form <Int><sp><Int>. e.g.: "3 3" is valid if the location (3,3) is empty.
    c. **Syntax of input**. As already mentioned, the syntax if move input is either 2 or 4 integers separated by spaces, but your program must allow for the user to append some space and a free form comment on the input line. For example:

    ```
    0 1     -- Black PlacedPawn
    ```

    is a valid input for a pawn placement. This requirement makes it easy to write scripted tests (see Note 1, below).

    d. **Syntax of prompts**: Prompts for moves must end in form "(B|W){1|2}:\n". That is, a B or a W to indicate player, immediately followed by a 1 or a 2 to indicate a pawn placement move (requiring one point specification [two integers]) or a normal move

(requiring two point specifications [four integers]). For example, a valid user prompt might be:

```
Enter the move coordinates for player White in
the form 'srcX srcY destX destY'
[0 >= n >= 4, or just enter return for a
'pass'] W2:

Enter the coordinates to place the pawn for
player Black in the form 'destX destY':
[0 >= n >= 4] B1:
```

This requirement is here to facilitate your program being run in a subprocess by another program analyzing your program's output and providing your program's input.

Non-Functional Requirements

1. **Haskell**. You must write your program in Haskell, and it must compile and run using GHC in versions 7.10.2-7.10.3).
2. **Use Lists for boards**. You MUST use a list of lists (`[ [ a ] ]`) as your primary data structure for representing the board. I have supplied a module, ApocTools.hs, which you MUST use, which will constrain your choice of data structures (and get you a head start on the assignment as well). I may test your use of this file by compiling your program with an alternate version (with the same interface) which will perform slightly different output related operations, or give me auxiliary trace data.
3. **ApocTools**. I will give you three files to get you started: Apoc.hs, a template for you assignment, and ApocTools.hs (updated 2016/02/07) a module to support the assignment, and ApocStrategyHuman.hs, an template for a strategy implementation (see also the haddock doc for these files (updated 2016/02/07)). You may NOT modify the file ApocTools.hs (even the file name) and you must link your program with this module. You are free to modify the Apoc.hs and ApocStrategyHuman.hs template files in any way you like: they are the bare-bones versions of your program files.
4. **Documentation**. Your source code should be well documented to the haddock standard. As part of the marking, we may run haddock on your source code to see the "haddock coverage" percentage.

Notes

1. **Echoing & testing**. When implementing the "human" strategy, you might find it convenient to echo each input line

so that's easier to follow in a trace. For example, the command line

```
   Apoc human human <
testBWins.txt > testBWins.log
```

will result in the a more readable file because it shows the input due to echoing the input lines whereas it would be hard to follow what was going on if it didn't.

2. **Repeating patterns**. Once you get around to implementing and testing your program, you will find that sometimes it goes into an endless loop. Commonly this is caused by two knights trying to take each other repeatedly. This is not acceptable. This problem is that knights have no "direction" and so easily fall into an endlessly repeating pattern. An easy way around this problem is to add a degree on randomness. For example, you could choose your best move only 90% of the time and randomly 10% of the time choose your second best move. This will break a repeating pattern (eventually) without dealing with the complex problem of pattern detection.
3. **Testing program**. (updated 2016/02/07) I provide you with a testing program, ApocCheckerMini.hs, which I highly recommend you use to check that your program will work with my testing program. This is a module used by my testing program, so the module name is not *main*. Therefore, you need to compile it explicitly specifying the main module like this:

   ```
   ghc
-main-is ApocCheckerMini ApocCheckerMini
   ```

   This program requires 6 external files (.in and .out files) that need to be in the default directory where you run ApocCheckerMini. You can download the whole package (including an updated ApocTools.hs file mentioned in *Non-Functional Requirements 3*) in this zip file. Note that this is not the test program I will use, but it's a scaled-down version with out the "smarts" to check your program dynamically -- it just compares your program's output to some output that my solution to the problem as produced. If I gave you complete test package, I'd be giving you the solution to the assignment.

Handing in this assignment

What to hand in? All I need is your source code. This will be, at minimum, your Apoc.hs file, but may include a small number of other module files in the same directory.
Where to hand it in? A D2L dropbox will be made available to submit this assignment.

Please note that this assignment will be partially marked by a program that runs your program as a subprocess. Therefore, if your program does not ***exactly*** follow these requirements the marking program will fail, and marks will be lost (maybe all your marks...) and/or you will be asked to fix your program so that it *does* conform to these requirements.