CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: April 15, 2003

Homework 8

Due: Friday, April 27, 2003.

This is the last and only two week homework problem set. Start early. It is not an assignment you can finish in a day or so.

Exercise 1 Recall the Torres game with which we started the class. This last assignment asks you to implement an automatic player for the game using the search techniques we have seen in class. Due to its complexity, we don't present the game in full generality, but restrict it to make it computationally feasible. Here are the rules.

Setup. The game consists of a 4×4 game board, is being played by two players and each player has three knights of one color. Furthermore, there is an almost infinite supply of game tiles that can be stacked in form of towers. A castle is defined as a collection of towers that touch each other at at least one edge.

Scoring. Each knight earns points according to height and surface area of the castle. The net worth of a castle is computed as the product of the maximal height of any player of one color on the board, and the surface of the castle. The surface of the castle is defined as the number of all towers on the board that are connected by sharing one edge. Two towers that only touch each other at a corner, need not be of the same castle.

Actions. There are three actions a player can undertake. Each action has a cost associated with it.

- Place a knight (next to an existing knight at the same or lower level). (2 credits)
- Move a knight from 1 square to another free square. During the move, the knight may move up at most one level, but can drop arbitrary many. Knights cannot move diagonally. (1 credit)
- Place a tile someplace on the board as an addition to an existing castle without connecting two castles (1 credit) *The current implementation does not check this feature yet, the tournament version, however will. Just make sure that you do not place the on an illegal position.*

Game: The game is played in 15 moves. Whoever has the most points wins.

In each move, each player has 2 credits to spend, and can put at most one tile on the board. The player does not have to use all of his or her credits. Leftover credits are counted as additional scoring points. During his or her move, a player can exercise any of the three actions described above.

Inital Board: Initially, the board is empty. The player who starts the game puts one of his tiles onto the board, with one of his or her knights on top of the tile. Then the opponent does the same thing.

The Torres game is described by the following signature. We use the standard board representation we studied already in Assignment 2, a relative way of addressing individual coordinates.

```
signature TORRES =
  sig
    datatype Color
      = White
      | Black

    datatype Tower
      = Empty
      | Piece of Color
      | Tile of Tower

    type Board

    datatype Address
      = Here
      | UpperLeft of Address
      | UpperRight of Address
      | LowerLeft of Address
      | LowerRight of Address

    datatype Direction
      = Up
      | Down
      | Left
      | Right

    datatype Action
      = Knight of Address
      | Move   of Address * Direction list
      | Place  of Address

    exception Invalid of string

    val displayBoard : Board -> unit
    val emptyBoard : Board
    val init : Color -> Board -> Address -> Board
    val action : Color -> Board -> Action list -> Board
    val inspect : Board -> Address -> Tower
  end
```

The course directory `/c/cs201/lib/ass8` contains many files, among them an imple-

mentation or the Torres game. Note, that you don't need to know what a board is. You are invited to look at the Torres structure in `torres.fun.sml` in the course directory, and take advantage of this particular way of implementing it, but you don't have to. You can design your own configuaration data structure, it does not have to be a "board". Your implementation is local to your player, nobody else can see it. You might want to think about extracting features from the board. For example, you might want to look for steps in the other castle you may want to try to climb...

The overall goal of this assignment that you implement the structure that stands for a player that satisfies the following signature.

```
signature PLAYER =
  sig
    val name : string
    val init : Torres.Color -> Torres.Board -> Torres.Address
    val move : Torres.Color -> Torres.Board -> Torres.Action list
  end
```

`name` should just return your name, `init` is called at the beginning of the game, and asks you where to put your initial tile an knight. Your opponent might have placed a tile and knight first, be sure to check that you don't jump on his head. If you do, you will be disqualified. And finally `move` takes a board as input and generates a list of actions that are to be executed. The "Color" that is passed in to `init` and `move` is the color you are playing. From this you can deduce the color of your opponent.

The course directory also contains a file called `sources.cfg` which defines all files that participate in this project. Copy all files int your personal working directory, and say `CM.make ()` which will load all the files in the right order — guaranteed.

The file `human.fun.sml` contains an interface to the Torres game for the human player. To see how it works, simply type in `Game.run ()` after you have loaded everything.

```
*-----------+-----------+-----------+-----------*
|           |           |           |           |
|-----------+-----------+-----------+-----------|
|           |#B         |           |           |
|-----------+-----------+-----------+-----------|
|           |           |#W         |           |
|-----------+-----------+-----------+-----------|
|           |           |           |           |
*-----------+-----------+-----------+-----------*
Alice>
```

There are two human players, Alice and Bob. First, Alice is asked to input something. The grammar for this interface is held intentionally simple. We will not use it very much, but it can give you a feeling for how to play this game. Type in "M13rr." which means "move knight at position 13 (which itself stands for `UpperLeft (LowerRight Here)`) two times to the the right (indicated by `r`). A sequence of moves must be terminated with a ".".

The result is

```
*-----------+-----------+-----------+-----------*
|           |           |           |           |
|-----------+-----------+-----------+-----------|
|           |#          |           |B          |
|-----------+-----------+-----------+-----------|
|           |           |#W         |           |
|-----------+-----------+-----------+-----------|
|           |           |           |           |
*-----------+-----------+-----------+-----------*
Bob>
```

The addressing is funny, but logical. "1" corresponds to `UpperLeft`, "2" to `UpperRight`, "3" to `LowerRight` and "4" to `LowerLeft`. For directions, "u" stands for `Up`, "r" for `Right`, "d" for `Down`, and "l" for `Left`.

Other commands include `P33`, place a piece into the lower right order, and `K34` for placing a knight below the white knight `W` on the board above. Arbitrary many of those actions can be written as one long string `P11P12`, the Torres game will make sure that you don't spend more points then allowed.

Now to the problems that you should work on. Recall, that all you have to do is to implement the structure

```
structure  Player_<name>  : PLAYER  =
  struct
    ...
  end
```

Warning: Don't just randomly create a list of actions as you are required to use min max search discussed in class.

- Develop your personal representation of configurations, and operations.
- Implement a function that assesses the quality of a configuration. State clearly what this function does. Your function should probably take into account the your current score vs. that of the opponent. Therefore write a function that computes the score for white and black.
- Implement

  ```
  val name : string
  ```

  just your name, for the protocol when we have the tournament.
- Implement

  ```
  val init : Torres.Color -> Torres.Board -> Torres.Address
  ```

  to give your initial choice for placing a tile and a knight. Look at the board. Pick a spot. Include some randomness here. Do not place your inital choice on top of your opponent.

- Implement

  ```
  val move : Torres.Color -> Torres.Board -> Torres.Action list
  ```

  Use a search algorithm based on min-max search with pruning, as discussed in class. This is the meat of the assignment.

- Each player will participate in the tournament and play each other. The tournament is organized as follows. The different players will be grouped into 8 groups. In each group everybody will play everybody. For the last eight, we will use an elimination scheme. The best three will receive a prize and extra credit points.

Do not edit the code from the course directory, because it is subject to change over the next few days.