Scrabble assignment

The project is to be done in groups of at least two or at most three (unless you are retaking the course. The same rules apply here as for the assignments.

This project revolves around creating a program capable of playing Scrabble according to the official scrabble tournament rules . Before you start you should familiarise yourself with these rules (they are not long at all) as the rest of the document assumes that you know them.

To help with the project we have created a Scrabble server that you will interact with and a scrabble bot of our own that you can compete against or just see how it operates. The server will keep an internal state of your game and ensure that all rules are followed. You will interact with the server using a given protocol. The server accepts a list of client functions and have these play against each other. This means that you can play against us, against your class mates, or against yourself.

We will make very precise and even provide code for some parts of this project, while leaving others more open-ended. To be more precise, the well-specified parts of this project are

- The protocols communicating with the server
- Setting up and maintaining a a connection to the server
- The DSL that defines the scrabble boards
- The rules that the game follows

The parts that are more open-ended are

- Your algorithm for determining the best possible moves
- How to maintain the scrabble board in memory after initialisation
- How to maintain a consistent internal state with that of the server (you will have all necessary information).

Finally, while we do adhere to the standard tournament rules for Scrabble we have also generalised our approach somewhat to make things a bit more interesting. The idea here is not to make things much harder, but rather provide a solid exercise in structuring your code in a general manner where game varieties can be created just by modifying some key parameters. More precisely, we allow

- Infinite boards
- Boards with holes in them
- Pieces that are placed on the board are represented of sets of characters and point values. The wild-card piece can then be seen as the set of all characters worth zero points, whereas a letter piece is a singleton set with that letter and its point value
- Tiles are functions that that operate on the word placed over them

We start by covering the types common to the server and the client, followed by the protocol used to communicate with the server and finally how to set up a communication with the server locally on your machine.

Before we start, however, one important note.

In this project we hope that you will share your compiled dlls with each other and compete and try to see how you fare against each other. We should not have to say this, but prior experience unfortunately says otherwise: **you may not share your source code with other groups**. If two groups hand in identical source code then both will be reported for plagiarism. To be on the safe side

1. Share ideas, not code. We have seen students excell at exams who have taken the time to explain concepts to their class mates rather than handing them code. Talking someone thorugh an algorithm on paper is great, giving them working code is a serious offence.
2. When passing around your dlls make sure to use `.fsi` files to hide everything except the one function that will interface with the server (this will be made clear later) You can also use the `internal` keyword to hide modules and types `module internal MyModule = ...` for instance. We will give you a top-level program that sets up the communication with the server and from here it is very easy to see what is visible or not.
3. There is code that accompanies this assignment that you may use.

This project is worth six points

- Two points for playing against yourself. You do not have to play well but you have to be able to stay in the game. Continuous passing is not playing the game ;). (Mandatory)
- Two points for playing against other people. This is really not that hard
- One point for parallelising the algorithm
- One point for writing an algorithm that respects the timeout flag

Playing the game

Before we check how everything is actually implemented you may want to run the small test engine that we have created. It allows you to play Scrabble against any bot that adheres to a specific API. You will be given skeleton code for your own project, but initially, you should try it out against `Oxyphenbutazone`, Jesper's scrabble bot.

To set up the game you need the following files which accompany the assignment.

- ScrabbleUtil.dll - contains the minimum required datatypes, a standard board to play on, and primitives for the server.
- ScrabbleServer.dll - Allows you to hook up an arbitrary amount of clients to play against each other, or several instances of the same client.
- Oxyphenbutazone.dll - Jesper's scrabble bot. It follows a greedy approach and always plays whatever the highest-scoring move it can find.
- Program.fs - Contains the code you will need to import other Scrabble bots and set up a game. This file will not change much but you may use it to modify the parameters of the games you play.
- EnglishDictionary.txt - This file should be placed at the root of your project. More precisely, it has the relative path `../../../EnglishDictionary.txt` to the compiled code.

To set this up, create a new Visual Studio project that runs in `VSCore 2.1` (Console App .NET Core). Once inside, do the following:

1. Update the NuGet packages
    - Upgarde FSharp.Core to version 4.7.1
    - Install FParsec (1.1.1)
    - Install FSharp.Quotations.Evaluator (2.1.0)
    - Install FsPickler (5.3.1) **not (5.3.2)**
2. Add `ScrabbleUtil.dll`, `ScrabbleServer.dll`, and `OxyphenButazon.dll` to the references of your project.
3. Add `Program.fs` and `EnglishDictionary.txt` to the root of your project.

The main function of `Program.fs` looks as follows:

```
let main argv =
    DebugPrint.debugFlag <- false // Change to true to enable debug output

    let board      = StandardBoard.standardBoard
    let alphabet   = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    let dictionary = readLines "../../../EnglishDictionary.txt"
    let handSize   = 7u
    let timeout    = None
    let tiles      = English.tiles 1u
    let seed       = None
    let port       = 13000
    let players    = spawnMultiples "Oxyphenbutazone"
Oxyphenbutazone.Startup.startGame 2
    // Uncomment this line to call your client
//  let players    = [("Your name here", YourClientName.Scrabble.startGame)]
    do ScrabbleServer.Comm.startGame
          board alphabet dictionary handSize timeout tiles seed port players

    0
```

At the top you find a mutable variable (*ducks*) that lets you enable or disable debug output. It is off by default, but by changig the flag to `true` you will see all of the attempted communications between client and server. This is highly useful to find bugs as you will be able to tell if someone is not receiving or sending when they should be. We also provide printing capabilities for you for debugging and **you should use these** rather than `printf` as you will otherwise clutter the console with debug output. Spamming `stdout` with debug information is not nice.

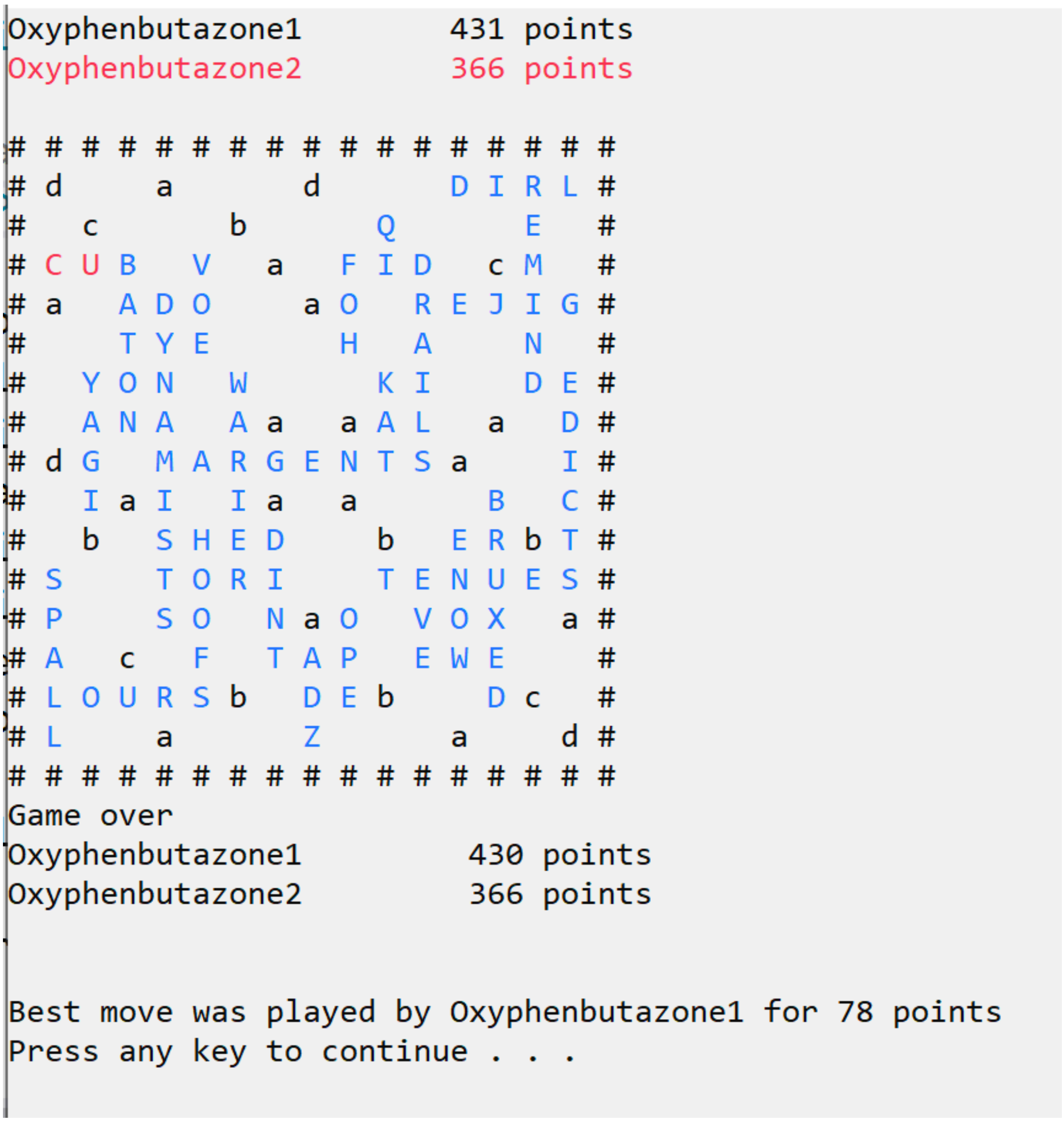
Moreover you have a `board` variable where you can choose which board to play on. We currently only have the stadard board but more will be added soon. You can create a dictionary as you did in Assignment 4 by using `alpabet` and `dictionary`. The `handsize` determines how many tiles you have on hand and `timeout`, which has type `uint32 option` determines if there is an upper limit to how long the client can think. The `tiles` tell us which letters we use (we will cover those later) and `seed` sets the random seed for the server which can be highly useful for debugging as you will get consistent behaviour. We communicate over TCP/IP and even though it's all on `localhost` you can set the portnumber by setting the `port` value. Finally, there is a list of players and by adding more client DLLs you can have them play against each other. In this particular example `Oxyphenbutazone` is playing against one copy of itself. There is also an alternative definition of `players` that you can use to hook up your own client, but we will get to that later.

If you run this you will start with something that looks like this (after some initial loading time)

```
# # # # # # # # # # # # # # # # #
# d     a       d       a     d #
#   c       b       b       c   #
#     c       a   a       c     #
# a     c       a       c     a #
#       c           c         #
#   b       b       b       b   #
#     a       a   a       a     #
# d     a               a     d #
#     a       a   a       a     #
#   b       b       b       b   #
#       c           c         #
# a     c       a       c     a #
#     c       a   a       c     #
#   c       b       b       c   #
# d     a       d       a     d #
# # # # # # # # # # # # # # # # #
```

This is an ascii representation of the board. Squares outside the board are marked with a `#`, single letter score tiles are marked with a space, double letter scores are marked with an `a`, triple letter scores are marked with a `b`, double word scores are marked with a `c`, and triple word scores are marked with a `d`.

After the game is done, it may look like this.



```
Oxyphenbutazone1        431 points
Oxyphenbutazone2        366 points

# # # # # # # # # # # # # # # # #
# d     a       d       D I R L #
#   c       b       Q       E   #
# C U B   V   a   F I D   c M   #
# a   A D O     a O   R E J I G #
#     T Y E       H   A     N   #
#   Y O N   W       K I     D E #
#   A N A   A a   a A L   a   D #
# d G   M A R G E N T S a     I #
#   I a I   I a   a       B   C #
#   b   S H E D     b   E R b T #
# S     T O R I     T E N U E S #
# P     S O   N a O   V O X   a #
# A   c   F   T A P   E W E     #
# L O U R S b   D E b     D c   #
# L     a       Z       a     d #
# # # # # # # # # # # # # # # # #
Game over
Oxyphenbutazone1         430 points
Oxyphenbutazone2         366 points


Best move was played by Oxyphenbutazone1 for 78 points
Press any key to continue . . .
```

As the game progresses, the placed letters are marked in red and the already existing letters are marked in blue. Debug information is displayed in magenta.

Setting up your own client

We have provided a code skeleton to set up your own solution to build off of. It manages all connections with the server and provides a means to manually input moves. You are given two files

1. Scrabble.fsi - An interface file containing the only function that needs to be exported to the server
2. Scrabble.fs - An implementation of the interface and the controll loop for the engine.

Create your own Library (not console application) running off of .NET Core (not .Net Standard) and give it a unique name to not clash with the other groups. Import Scrabble.fs and Scrabble.fsi and your MultiSet implementation from Assignment 4. Import your library into the project you already have, swap out `Oxyphenbutazone` for your new project and you should see the following:

```
# # # # # # # # # # # # # # # # #
# d     a       d       a     d #
#   c     b           b     c   #
#     c       a   a       c     #
# a     c       a       c     a #
#   b     b           b     b   #
#                               #
#     a       a   a       a     #
# d     a               a     d #
#     a       a   a       a     #
#                               #
#   b     b           b     b   #
# a     c       a       c     a #
#     c       a   a       c     #
#   c     b           b     c   #
# d     a       d       a     d #
# # # # # # # # # # # # # # # # #




0 -> (set
  [('A', 0); ('B', 0); ('C', 0); ('D', 0); ('E', 0); ('F', 0); ('G', 0);
   ('H', 0); ('I', 0); ...], 1)
1 -> (set [('A', 1)], 1)
4 -> (set [('D', 2)], 1)
5 -> (set [('E', 1)], 2)
9 -> (set [('I', 1)], 1)
16 -> (set [('P', 3)], 1)
Input move (format '(<x-coordinate><y-coordinate> <piece id><character><point-
value> )*', note the absence of state between the last inputs)
```

The board is displayed just like above.

Moreover your hand is displayed. Each tile is represented by a set of possible characters that it can bi enstantiated to. In this example, all letters are singleton sets but the top one is the wildcard tile that can be instantiated with any letter. You will always have seven pieces on hand until the server runs out, and the line `4 -> (set [('D', 2)], 1)` means that you have one piece where the character `D` worth `2` points is a valid instantiation (remember that you can have sets of characters which the first line demonstrates) and that the unique id for this tile is `4`.

To play a word you type in the letters that you want to play on one line in the following format `<x-coordinate> <y-coordinate> <tile id><character><point-value>`. For instance, if you wanted to write the word `DO` on the board starting at the centre and going down you would write

```
0 0 4D2 0 1 0O0
```

where the second tile makes use of the wild-card character. At this moment the internal game state is updated, but we have not updated the local state which should be one of the first things you do. We will talk more about state soon. But once you get this set up you have something to work off of.

Types

In order for communication to work between client and server there are some types that we must agree on. This section covers these. All types can be found in **ScrabbleUtil.dll**.

Coordinate system

We work in a discrete two-dimensional coordinate space. Since boards can be infinite in all directions coordinates are given as pairs of integers

`type coord = int * int`

The center of the board is typically at `(0, 0)` but it does not have to be.

Boards

The type for boards looks as follows:

```
type boardProg = {
        prog       : string;
        squares    : Map<int, squareProg>
        usedSquare : int
        center     : coord

        isInfinite : bool   // For pretty-printing purposes only
        ppSquare   : string // For pretty-printing purposes only
    }
```

Boards are represented using the DSL that we created before. The program `prog` will have two variables `_x_` and `_y_` as arguments which represents the coordinates and will store an integer in a variable called `_result_`. This integer is a unique identifier for the square at this coordinate and the program for that square can be obtained from the `squares` lookup table. If the number is not in the `squares` then that part of the board is blank (we do support boards with holes in them).

Boards also have a `center` coordinate over which the first wourd must be placed.

There is also a `usedSquare` field which is also covered in the next section, but the intuition is that we use this square to use point scores for letters that have already been placed on the board -- if a piece is placed over a Double Letter Score square, for instance, then that square cannot be used again but the point value of the piece itself can. In effect placing a piece on a square changes the square to something else, but it's the same square for all used pieces.

There are also some fields for pretty-printing but all of that is handled by the server.

Squares

Squares have the following type

`type squareProg = Map<int, string>`

In Assignment 2 we worked on how to calculate the point values generated from squares where every square had a set of functions of varying priority. Losely speaking when we place a word over a set of squares we collect all functions that comprise those squares, sort them by priority, compose them, and calculate the points we get. You have already done this part in Assignment 2, and you have already transformed the AST of our DSL to functions that Assignment 2 can handle in Assignment 6.

What remains to do is to compile every program in the map to a function of the type

`index : uint32 -> word : (char * int) list -> points : int -> int`,

where `word` is a list of letters and their point values that forms the word that has just been placed over the tile (`[|('Q', 10); ('I', 1); ('N', 1)|]` for instance), `index` is the index into this list that points to the exact letter that is placed over this particular tile (`0` for `('Q', 10)`, `1` for `('I', 1)`, and `2` for `('N', 1)`), and finally `points` refers the number of points that we have calculated so far when calculating the total score that a word gives us.

We will do an example shortly, but bear in mind that by making our tiles this general we are very flexible in how our scores are calculated.

Another thing to note is that a tile can never have two functions of the same priority, but adjacent tiles can and this becomes useful when calculating the score for entire words. This does mean, however, that we cannot be reliant on the order in which we perform functions of the same priority and we therefor guarantee that all such functions are commutative (an operator `op` is commutative if `a op b` is equal to `b op a`).

For squares, we use the variable `_pos_` as an argument for where we are in the word, `_acc_` to keep track of how many points we have so far and `_result_` to store the return value.

As an example for a tile, consider the Tripple Letter Score

```
let trippleLetterScore : squareProg =
    Map.add 0 "_result_ := pointValue(_pos_) * 3 + _acc_" Map.empty
```

This function has priority `0` meaning that it will it will be among the first to run and it takes the current point value `_acc_` and adds it to the point value of the letter placed over the tile at position `_pos_` multiplied by 3. We also have similar tiles for single- and double letter scores.

The Double Word tile, on the other hand needs to run after all the points for the individual pieces have been collected and only multiplies the points that have been accumulated so far.

`let doubleWordScore = Map.add 1 "_result_ := _acc_ * 2" singleLetterScore`

You can use the code you did in Assignment 2 to calculate the points.

Server communication

Setting up the communication between the client and the server is handled for you, but we will cover it briefly.

Your scrabble client will expose a single function

```
val startGame :
   boardProg ->                     (* Scrabble board *)
   string ->                        (* Alphabet *)
   string list ->                   (* Dictionary *)
   uint32 ->                        (* Number of players *)
   uint32 ->                        (* Your player number *)
   uint32 ->                        (* starting player number *)
   (uint32 * uint32) list ->        (* starting hand (tile id, number of tiles) *)
   Map<uint32, tile> ->             (* Tile lookup table *)
   uint32 option ->                 (* Timeout in miliseconds *)
   Stream ->                        (* Communication channel to the server *)
   (unit -> unit)                   (* Delay to allow everyone to start at the
                                       same time after startup *)
```

It takes as argument everything you need to set up the game. One thing to pay attention to is that there is a lookup table for the tiles whereas your hand is a list of pairs of integers where the first number is the id of the tile, and the second is how many you have. There is also a communication channel with the server, but all of the initialisation is handled on the server's end.

The server, on the other hand, exposes the following function which we have already called form `Program.fs`.

```
  val startGame :
       boardProg ->              (* Board *)
       string ->                 (* alphabet *)
       string list ->            (* dictionary *)
       uint32 ->                 (* hand size *)
       uint32 option ->          (* timeout *)
       (tile * uint32) list ->   (* tiles *)
       int option ->             (* random seed *)
       int ->                    (* port *)
       (string *
        (boardProg -> string -> string list ->
           uint32 -> uint32 -> uint32 -> (uint32 * uint32) list ->
Map<uint32, tile> ->
           uint32 option -> NetworkStream -> (unit -> unit))) list ->
       unit
```

The first arguments are what is required to set up the game and the final list is a list of clients that follow the specification above. Once the game is started, you can communicate with the server using the following messages. which will be explained in detail below but are heref or the sake of completion.

```
    type ServerMessage =
        | SMPlay      of (coord * (uint32 * (char * int))) list
        | SMPass
        | SMForfeit
        | SMChange    of uint32 list

    type ClientMessage =
        | CMPlayed             of uint32 * (coord * (uint32 * (char * int)))
list * int
        | CMPlaySuccess        of (coord * (uint32 * (char * int))) list *
                                   int * (uint32 * uint32) list
        | CMPlayFailed         of uint32 * (coord * (uint32 * (char * int)))
list
        | CMPassed             of uint32
        | CMForfeit            of uint32
        | CMChange             of uint32 * uint32
        | CMChangeSuccess      of (uint32 * uint32) list
        | CMTimeout            of uint32
        | CMGameOver           of (uint32 * int) list

    type GameplayError =
    | GPEOccupiedTile of (char * int) * coord * (char * int)
    | GPEWordNotOnRowOrColumn of coord list
    | GPEEmptyMove
    | GPEInvalidPieceInst of uint32 * (char * int)
    | GPEPieceDoesNotExist of uint32
    | GPEEmptyTile of coord
    | GPEPlayerDoesNotHavePiece of uint32 * uint32
    | GPEWordNotConnected
    | GPEWordOutsideBoard
    | GPEWordNotInDictionary of string
    | GPEFirstWordNotOverCenter of coord
    | GPEFirstWordTooShort
    | GPENotEnoughPieces of uint32 * uint32
    | GPEWordNotAdjacent
```

Messages to the server

There are only four things you can do to the server - playing a move, passing, swapping pieces, and resigning.

Playing a move

Name:	`SMPlay tiles`
Type:	`(coord * (uint32 * (char * int))) list -> ServerMessage`

Attempts to place the tiles in the list `tiles` on the board. The elements of the list has the form `(coordinate : coord, (tileId : uint32, tileInstantiation : (char * int)))`. The term `coordinate` is where an individual tile is placed, the term `tileId` is the id-number of the tile being placed, and `tileInstantiation` is a valid instantiation of that piece (character and score). The coordinates **do not** have to be ordered in any special way. The server will figure out placement on its own.

On success

- The message `CMPlaySuccess` is sent to the player who made the move
- The message `CMPlayed` is sent to all other players

On Failure

- A lints of gameplay errors are returned to the player
- The message `CMPlayFailed` is sent to all other players

If a move fails, that player's turn is over and the turn goes to the next player.

Possible errors

- `GPEOccupiedTile` - The player tried to place a piece on an occupied tile
- `GPEWordNotOnRowOrColumn` - The player attempted to place pieces that were not along a single row or column on the board
- `GPEEmptyMove` - The player tried to make an empty move (in effect this will work like passing since turn goes over to the next player)
- `GPEInvalidPieceInst` - The player tried to instantiate a piece with id `pieceId` with an invalid instantiation `pieceInstantiation`
- `GPEPieceDoesNotExist` - The player attempted to place a piece with id `pieceId` which does not exist in the collection
- `GPEEmptyTile` - The player attempted to place a piece outside of the board
- `GPEPlayerDoesNotHavePiece` - The player attempted to place a piece that they do not own
- `GPEWordNotConnected` - The pieces were placed along one row or column, but they did not form one cohesive word with pieces already on the board
- `GPEWordOutsideBoard` - Part of the word landed outside of the board
- `GPEWordNotInDictionary` - One of the words formed was not in the dictionary
- `GPEFirstWordNotOverCenter` - The player tried to place a word on an empty board where the word did not cross the center coordinate
  *`GPEFirstWordTooShort` - The player tried to play a word on an empty board that was less than two characters long

* `GPENotAdjacent` - The player tried to place a word on a non-empty board that was not adjacent to an already existing word

Passing

Name:	`SMPass`
Type:	`ServerMessage`

This function always succeeds and the message `CMPassed` is broadcast to all players. If all players pass for three consecutive turns the game ends.

Forfeiting the game

Name:	`SMForfeit`
Type:	`ServerMessage`

This function always succeeds and the message `CMForfeit` is broadcast to all players. This player will no longer be a part of the player list and it is therefor important that the clients keep track of who is still playing so that they do not wait for people who are no longer in the game.

Change pieces

Name:	`SMChange`
Type:	`uint32 list -> ServerMessage`

A player attempts to swap tiles form their hand. If successful, the server will send back new tiles. Changing pieces ends your turn.

On success

The server sends `CMChangeSuccess` to the player changing pieces and broadcasts `SMChange` to everyone else.

On failure

- `GPENotEnoughPieces` - There player tried to switch more pieces than there are free pieces left in the game

- `GPEPlayerDoesNotHavePiece` - The player tried to change a piece they do not possess
- `GPEPieceDoesNotExist` - The player tried to change a piece that does not exist

Messages to the client

These messages are used to communicate game state changes to the players so that they can keep up to date.

Successful play by other player

Name:	`CMPlayed(playerId, move, points)`
Type:	`uint32 * (coord * (uint32 * (char * int))) * int -> ClientMessage`

The player `playerId` successfully played the pieces `move` (which is on the same format as in `SMMove`) and received `points` points.

Successful play by you

Name:	`CMPlaySuccess(move, points,newTiles)`
Type:	`(coord * (uint32 * (char * int))) * int * (uint32 * uint32) list -> ClientMessage`

You successfully played the tiles `move` (which is on the same format as in `SMMove`), received `points` points, and the new tiles `newTiles` (again on the same format as your initial hand in `SMMove`) to replace the ones you played.

Failed play

Name:	`CMPlayFailed(playerId, move)`
Type:	`uint32 * (coord * (uint32 * (char * int))) -> ClientMessage`

The player `playerId` failed to play the tiles `move` (which is on the same format as in `SMMove`).

Player passed

Name:	`CMPassed(playerId)`
Type:	`uint32 -> ClientMessage`

The player `playerId` passed

Player forfeit

Name:	`CMForfeit(playerId)`
Type:	`uint32 -> ClientMessage`

The player `playerId` left the game

Other player successfully changed pieces

Name:	`CMChange(playerId, numberOfTiles)`
Type:	`uint32 * uint32 -> ClientMessage`

The player `playerId` successfully changed `numberOfTiles` tiles

You successfully changed pieces

Name:	`CMChangeSuccess(newTiles)`
Type:	`(uint32 * uint32) list-> ClientMessage`

You successfully changed pieces and received `newPieces` to replace the ones you changed (on the same format as for `SMSend`).

Player timeout

Name:	`CMTimeOut(playerId)`
Type:	`uint32 -> ClientMessage`

The player `playerId` timed out. This counts as passing for all gameplay purposes.

Game over

Name:	`CMGameOver(finalScore)`
Type:	`(uint32 * int list) -> ClientMessage`

The game is over and a list of player identifiers and their final score is returned.

Gameplay errors

The following errors are used whenever invalid moves are attempted by the players. We will use the name `instantiation` to mean things of type `char * int`, i.e. a letter that has actually been placed (or is about to be placed) on the board.

Placing piece on occupied square

Name:	`GPEOccupiedTile(instantiation, coordinate, currentInstantiation)`
Type:	`(char * int) * coord * (char * int) -> GameplayError`

You attempted to place `instantiation` on the coordinate `coordinate` but the letter `currentInstantiation` was already placed there.

Pieces not placed along a row or column

Name:	`GPEWordNotOnRowOrColumn(coordinates)`
Type:	`coordinate list -> GameplayError`

Tried to place pieces on the coordinates `coordinates` that were not along a single row or column.

Empty move

Name:	`GPEWordNotOnRowOrColumn`
Type:	`GameplayError`

You played a move without any pieces.

Invalid tile instantiation

Name:	`GPEInvalidPieceInst(tileId, instantiation)`
Type:	`uint32 * (char * int) -> GameplayError`

You tried to instantiate the tile `pieceId` with an invalid instantiation `instantiation`. This happens if your instantiation is not a member of the set of valid instantiations of the piece represented by `tileId`

Trying to place a tile that does not exist

Name:	`GPEPiecedoesNotExist(tileId)`
Type:	`uint32 -> GameplayError`

You tried to use a piece with id `tileID` does not exist.

Trying to place tile on empty square

Name:	`GPEEmptyTile(coordinate)`
Type:	`coordinate -> GameplayError`

You tried to place a tile on an empty square at coordinate `coordinate`

Placing a tile that you do not have

Name:	`GPEPlayerDoesNotHavePiece(playerId, tileId)`
Type:	`uint32 * uint32 -> GameplayError`

You (player `playerId`) tried to use a piece with id `tileId` that you do not have.

Placing a word that is not connected

Name:	`GPEWordNotConnected`
Type:	`GameplayError`

You tried to place a word that was on a single row or column, but the pieces did not form one cohesive word with the pieces already on the board.

Placing a word partially outside the board

Name:	`GPEWordOutsideBoard`
Type:	`GameplayError`

You tried to place a word that was at least partially outside the board.

Word not in dictionary

Name:	`GPEWordNotInDictionary(word)`
Type:	`string -> GameplayError`

The pieces you placed formed a word `word` that is not in the dictionary.

First word is not over the center

Name:	`GPEFirstWordNotOverCenter(coordinate)`
Type:	`coord -> GameplayError`

As the first move of the game you tried to place a word that was not over the center coordinate `coordinate`.

First word is too short

Name:	`GPEWordTooShort`
Type:	`GameplayError`

As the first move of the game you tried to place a word that was not at least 2 characters long.

Not enough tiles

Name:	`GPENotEnoughPieces(changeTiles, availableTiles)`
Type:	`uint32 * uint32 -> GameplayError`

You tried to change `changeTiles` number of tiles, but there are only `availableTiles` number of tiles left in the game.

Word placed is not adjacent to another one

Name:	`GPEWordNotAdjacent`
Type:	`GameplayError`

You tried to place a word that is not adjacent to any other words on the board.

Server setup

To work against the server you need to keep your own local state, which runs in sync with the server, and a means to open and maintain data connections to the server.

State

To play the game you will have to maintain the state of the current game. Exactly what this is is up to you, but we highly recommend that you keep it in a record structure of your own. For our example program we have used the following data structure

```
type state = {
    playerNumber  : uint32
    hand          : MultiSet.MultiSet<uint32>
}
```

The state record contains two fields `playerNumber` and `hand`, where `playerNumber` is your player id. The field `hand` contains the current tiles you have on hand as a multiset of integers. In combination with the piece map from the previous section you can ensure that you know which concrete pieces you have on hand.

Other things that you need to keep track of is who's turn it is so that you can act when it's your turn, but this list is by no means exhaustive, but enough to get you started and enough to create a scrabble-bot that only plays against itself.

Communicating with the server

We are providing the code that sets up the game for you so that all you have to worry about is the logic of the actual game. You communicate to the server via a `Stream` using TCP/IP. A skeleton game loop can be found here.

```
        let rec aux (st : State.state) =
            Print.printHand pieces (State.hand st)

            forcePrint "Input move (format '(<x-coordinate> <y-coordinate>
<piece id><character><point-value> )*', note the absence of state between the
last inputs)\n\n"
            let input =  System.Console.ReadLine()
            let move = RegEx.parseMove input

            debugPrint (sprintf "Player %d -> Server:\n%A\n"
(State.playerNumber st) move)
            send cstream (SMPlay move)

            let msg = recv cstream
            debugPrint (sprintf "Player %d <- Server:\n%A\n"
(State.playerNumber st) move)

            match msg with
            | RCM (CMPlaySuccess(ms, points, newPieces)) ->
                (* Successful play by you. Update your state
                   (remove old tiles, add the new ones, change turn, etc) *)
                let st' = st // This state needs to be updated
                aux st'
            | RCM (CMPlayed (pid, ms, points)) ->
                (* Successful play by other player. Update your state *)
                let st' = st // This state needs to be updated
                aux st'
            | RCM (CMPlayFailed (pid, ms)) ->
                (* Failed play. Update your state *)
                let st' = st // This state needs to be updated
                aux st'
            | RCM (CMGameOver _) -> ()
            | RCM a -> failwith (sprintf "not implmented: %A" a)
            | RGPE err -> printfn "Gameplay Error:\n%A" err; aux st


        aux st
```

We use the functions `send` and `recv` from `ScrabbleUtil` to communicate with the server. We can then match on the messages we receive, update our state accordingly, and recurse.

Note the use of `debugPrint`. This function allows you to print in a multi-threaded environment. We highly recommend that you use these rather than `printf` as they can be switched off easily when they are no longer needed.

Parallelism and interrupts

Parallelising a scrabble engine is in principle close to trivial as very few of the computations depend on previous one.

- Building a word from one requires no information from words built on other tiles
- Using one piece from your hand on a tile to continue a word can be done in parallel with choosing another letter from your hand and continue another word
- Using one element from one the piece sets can be done in parallel with choosing another element from the same set.

In reality, however, things are not quite as easy. Setting up parallelism produces overhead and if you were to parallelise all of the steps mentioned above then you would end up with an algorithm that is slower than the linear one. You will have to experiment a bit to see what sticks.

Options for parallelism

When it comes to parallelism you have two choices. The first is to use the `Async` framework to fork of asynchronous processors and collect the data. For instance, if you have a function `doAction : input -> Async<result>` then you can, given input, obtain an asynchronous process that calculates the result. You can then use `Async.Parallel : Async<‘a> list -> Async<‘a[]>` to do the computation. As an example, assume you have a list `actions : Async<result>` then you can do the following at the top-level.

```
actions |>
Async.Parallel |>
Async.RunSynchronously
```

This command will then return an array of results that you can collect by folding over the array.

Another (and most likely simpler) option is to use `System.Threading.Tasks.Parallel.ForEach : IEnumerable<‘a> -> (‘a -> unit) -> ParallelLoopResult`

This can, for instance be set up in the following way (assuming that you have a function `doAction : input -> ()`).

```
open System.Threading.Tasks
Parallel.ForEach (inputSet, doAction) |> ignore
```

Since this action does not return a result you will need to store the information somehow, and this can be used by a mutable variable. You do, however, have to be careful as you can get race conditions on this variable. By far the easiest way to handle this is by using mailboxes. It’s short, succinct, and elegant and described in depth here.

Using mailboxes it is safe to have a mutable field that stores your best move so far as the mailboxes will make sure that changes to this field are only applied in order. The best (functional) way to do it is, however, to have two types of messages that you can put in your mailbox — one message that puts things in the mailbox, and one message containing a continuation that takes a message out of the mailbox and passes it to the continuation. You are, however, free to use mutable data if you wish.

Another option is to use locks, but this is really more error-prone and more complicated. We highly recommend you use the mailboxes.

Cancelling actions

We have timeouts in the game as otherwise we would not be able to handle that people drop out of the game for whatever reason (buggy code, internet malfunction, ...). Timeouts are handled in .NET using something called cancellation tokens. These can be used to cancel ongoing tasks. For Async workflows, for instance, you can use `Async.Start : Async<unit> * CancellationToken -> unit` that starts an asynchronous task but that can be cancelled using a cancellation token. For more information read this post.

You can also cancel `Parallel.ForEach` with cancellation tokens. The following code sets up a task that will be cancelled automatically after a specific timeout.

```
use cts = new System.Threading.CancellationTokenSource(timeoutInMiliseconds)
let po = new ParallelOptions()
po.CancellationToken <- cts.Token
po.MaxDegreeOfParallelism <- System.Environment.ProcessorCount

try
    Parallel.ForEach (inputSet, po, doAction) |> ignore
with
| :? System.OperationCanceledException -> printfn "Timeout"
```

Note that whenever the operation is cancelled an exception is thrown that you will have to catch and then return the best move that you have discovered so far (again using mutable variables and/or mailboxes).