The project is to be done in groups of at least two or at most three (unless you are retaking the course.

This project has you 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 and 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 clients 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 some parts of the project very precise and even provide code for some parts, while leaving other parts 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 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 operate on the words placed over them

Before we start, however, one important note.

In this project we hope that you will compile DLLs of your bots and send them around to other groups to test againtst each other. It should go without saying, but prior experience unfortunately says otherwise: **you may not share your source code with other groups**. If two groups hand in identical or similar enough source code then both will be reported for plagiarism. To be on the safe side

1. Share ideas, not code. We have seen students excel at exams who have taken the time to explain concepts to their class mates rather than handing them code. Talking someone through an algorithm on paper is great, giving them working code is a serious offence.
2. When compiling your client make sure that all `.fsi` files 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 = ...` and `type internal myType = ...` 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. Any code provided by us for the project you may use and hand in without referencing us a source.

This project is worth six points

- Two points for playing against yourself on the infinite board without holes. 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)
- One point for playing against other people and implementing a Trie or a Gaddag as your dictionary
- One point for being able to finish a game on all boards using the DSL either by using your own parser (assignmnts 6 and 7) or using the slimed down version that we provide (covered later).
- One point for parallelising the algorithm
- One point for writing an algorithm that respects the timeout flag

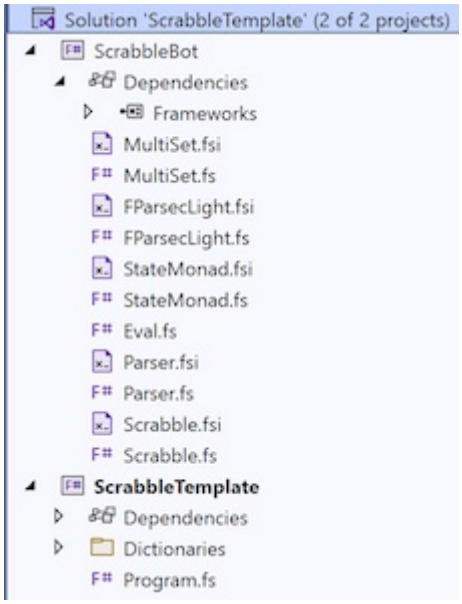
Setting up your project

In this section we will cover how to set up your project for the first time.

You will get a Visual Studio solution containing two projects -- one library `ScrabbleBot`, which will be your scrabble bot, and one executable `ScrabbleTemplate` that can load existing bots and have them play against each other.

Important: This project will build, but will throw exceptions when you run it until you have imported the relevant parts from your assignment.

The `ScrabbleTemplate` solution that you will work off of looks as follows.



In addition, these projects depend on four dlls packages

- `ScrabbleServer` - Allows you to hook up an arbitrary amount of clients to play against each other, or several instances of the same client to play against itself. * `ScrabbleUtil` - a utility library that contains the minimum required datatypes, a few boards to play on, and primitives for server communication, and a means to set up a common dictionary among several bots (you only have to implement one for yourself)
- `ScrabbleLib` - A shared library to support the other DLLs. The only function available to you is a simple parser for the scrabble boards (covered later)
- `Oxyphenbutazone` - Jesper's scrabble bot. It follows a greedy approach and always plays the highest-scoring move it can find.

To get started need to Import your solutions from previous assignments to the `ScrabbleBot` project.

Getting Started

Before we start we will have a brief look at the main program in `Program.fs` that sets up communication between the clients. This will give you a high-level view of how the system functions. We go into detail more later.

```
let main argv =
    ScrabbleUtil.DebugPrint.toggleDebugPrint true // Change to false to
supress debug output
```

```
    System.Console.BackgroundColor <- System.ConsoleColor.White
    System.Console.ForegroundColor <- System.ConsoleColor.Black
    System.Console.Clear()

    let board        = ScrabbleUtil.StandardBoard.standardBoard ()
//    let board      = ScrabbleUtil.InfiniteBoard.infiniteBoard ()
//    let board      = ScrabbleUtil.RandomBoard.randomBoard ()
//    let board      = ScrabbleUtil.RandomBoard.randomBoardSeed (Some 42)
//    let board      =
ScrabbleUtil.InfiniteRandomBoard.infiniteRandomBoard ()
//    let board      =
ScrabbleUtil.InfiniteRandomBoard.infiniteRandomBoardSeed (Some 42)
//    let board      = ScrabbleUtil.HoleBoard.holeBoard ()
//    let board      = ScrabbleUtil.InfiniteHoleBoard.infiniteHoleBoard ()



    let words     = readLines "../../../Dictionaries/English.txt"
    let handSize   = 7u
    let timeout    = None
    let tiles      = ScrabbleUtil.English.tiles 1u
    let seed       = None
    let port       = 13001



    let dictAPI =
        // Uncomment if you have implemented a dictionary. last element
None if you have not implemented a GADDAG
        // Some (Dictionary.empty, Dictionary.insert, Dictionary.step,
Some Dictionary.reverse)
        None

    let (dictionary, time) =
        time (fun () -> ScrabbleUtil.Dictionary.mkDict words dictAPI)

    // Uncomment to test your dictionary
    // ScrabbleUtil.DebugPrint.debugPrint ("Dictionary test sucessful\n")
    // let incorrectWords = ScrabbleUtil.Dictionary.test words 10
(dictionary false) // change the boolean to true if using a GADDAG
    // match incorrectWords with
    // | [] -> ScrabbleUtil.DebugPrint.debugPrint ("Dictionary test
sucessful!\n")
    // | _  ->
    //    ScrabbleUtil.DebugPrint.debugPrint ("Dictionary test failed for
at least the following words: \n")
    //    List.iter (fun str -> ScrabbleUtil.DebugPrint.debugPrint
(sprintf "%s\n" str)) incorrectWords
```

```
    // Uncomment this line to call your client
    // let players    = [("Your name here", dictionary,
YourClientName.Scrabble.startGame)]

    let players = spawnMultiples "OxyphenButazone" dictionary
Oxyphenbutazone.Scrabble.startGame 2

    do ScrabbleServer.Comm.startGame
          board dictionary handSize timeout tiles seed port players
    ScrabbleUtil.DebugPrint.forcePrint ("Server has terminated. Press
Enter to exit program.\n")

    System.Console.ReadLine () |> ignore

    0
```

This file allows the user to set up the board being played on, the dictionary being used, the size of the hand of the players, the timeout (if any), the tiles you will be playing with and a random seed (if any) which is useful for debugging as the behaviour of the program becomes more deterministic. There is also a list `players` which is the list of bots that will be playing against each other. As a default it is set up to pair two versions of Jesper's bot Oxyphenbutazone against themselves.

The function `ScrabbleUtil.Dictionary.mkDict` takes the dictionary that you have implemented in Assigment 4 (either a Trie or a Gaddag), by initialising it with your functions, and returns a wrapper for that dictionary so that your client, other clients, and the server can use it. The reason for this is that building dictionaries takes time and if every client builds their own then we literally can have to wait minutes before we start depending on how efficient your implementations are. If you have not implemented a dictionary then setting the `dictAPI` variable to `None` provides you with a dictionary that can only recognise words of odd length.

The funtion `ScrabbleUtil.Dictionary.test` tests that your dictionary is equivalent to the one that Oxyphenbutazone uses. This test takes time and is only there for your debugging purposes.

Finally, the function `ScrabbleServer.Comm.startGame` starts a game of Scrabble with all of the settings that you have done.

We will come back to this in more detail later, but for now, we cover how to set up the project.

Running the project

You should now be able to run two instances of Jesper's bot Oxyphenbutazone against each other. After a completed game, the board can look like this (note that

only words of odd length have been played since you have not provided a dictionary yet):

```
OxyphenButazone1         334 points
OxyphenButazone2         249 points

# # # # # # # # # # # # # # # # # #
# d     a       d     D J I N S #
#   C   U   b       H   O   c   #
#   R c N   R a   P E R I L     #
# a O   G   E   B   N   N     F #
#   F O L I A   A   R U T     I #
#   T   U   W I N D Y   U   b T #
#     a E   O a   a     R a   L #
# d     D   K O R     M E A T Y #
#     a     E a E a T   S a A   #
#   b       b   D A W       V   #
# A     M A N G O   I c         #
# B     O       X E N I C     a #
# I   c A     Z   a G   I c     #
# D E P T H b O     E D G E D   #
# E     S   S A V E D   a     d #
# # # # # # # # # # # # # # # # # #
Player 2 <- Server :
RCM (CMPlayed (1u, [((4, 5), (9u, ('I', 1)))], 6))
Player 2 -> Server :
SMPass
Server -> Player 2 :
RCM (CMGameOver [(2u, 238); (1u, 334)])
Server -> Player 1 :
RCM (CMGameOver [(2u, 238); (1u, 334)])
Player 2 <- Server :
RCM (CMGameOver [(2u, 238); (1u, 334)])
Listener <- Server      :
RCM (CMGameOver [(2u, 238); (1u, 334)])
Player 1 <- Server :
RCM (CMGameOver [(2u, 238); (1u, 334)])
Game over
OxyphenButazone2         238 points
OxyphenButazone1         334 points


Best move was played by OxyphenButazone1 for 40 points
Server has terminated. Press Enter to exit program.
```

It will print the entire game with debug output (that you can switch off if you wish). In this game `Oxyphenbutazone1` won the game with 334 points to 238 and played the best move for 40 points.

Import your assignments

You need to get your assignments imported into the project. For all of these you need to be a bit careful about your namespaces in case you have renamed anything from your assignments. Look at the placeholder `.fsi` files to see what should be there.

The project compiles from the start. Add a little bit at a time and make sure that it keeps compiling as you add your assignments as it can be difficult to find a bug otherwise.

First of all, make sure that all of your modules and types, except for the ones in `Scrabble.fs` and `Scrabble.fsi` are internal so that they are not exported when you compile your bot That is, all `module Name ...` should be `module internal Name ...` and any type outside of a module (inside is fine as they will be hidden by the module itself) of the form `type name ...` should be changed to `type internal name ...`.

Secondly,

- In `Program.fs` in the `ScrabbleTemplate` project, find the `main` function and the following code

```F#
  let dictAPI =
      // Uncomment if you have implemented a dictionary.
      // last element None if you have not implemented a GADDAG
      // Some (Dictionary.empty, Dictionary.insert, Dictionary.step,
      //.      Some Dictionary.reverse)
      None
   let (dictionary, time) =
      time (fun () -> ScrabbleUtil.Dictionary.mkDict words dictAPI)
```

Leave this code as it is if you have not implemented a dictionary, otherwise comment out the `None` expression, uncomment the commented out rows and add references to your dictionary (that you should preferably place in a module called Dictionary). If you

have implemented a GADDAG then the last argument sholud be `Some Dictionary.reverse` and `None` otherwise.

Finally, you have to set up your own bot.

In the project `ScrabbleBot`, while maintaining the module structure (keep the module names from the `.fsi` files and make sure they are internal)

- Replace `MultiSet.fsi` and `MultiSet.fs` with your solution from Assignment 4
- Replace `StateMonad.fs` with your solution from Assignment 6. `StateMonad.fsi` is the same as before and can remain the same.
- Replace `Eval.fs` with your solution from Assignment 6
- Replace `Parser.fs` with your solution from Assignment 7. Make sure that:

 * It uses `FParsecLight` and not `JParsec`. The interfaces are identical. You should not have to alter a single line of code, except for the `open` statement, in your program to make this switch.

 * Remove the `Ass7` module and merge it with `ImpParser` (look at `Parser.fsi` for details)

 * Add the line `open ScrabbleUtil` to the `Parser.fs`.

 * The types `boardProg`, `coord`, and `squareProg` are defined in the `ScrabbleUtil` library. Remove them from the `Types.fs` file (or remove the file completely).

- In `Scrabble.fsi` and `Scrabble.fs`

 * change the namespace name to your `ProjectId` that you used for your `ScrabbleBot.fsproj` file. You should have a unique bot name here for the final tournament.

 * In the function `startGame` there are two rows where one is commented out depending on wether or not you have used a Gaddag.

 ```F#
 //let dict = dictf true // Uncomment if using a gaddag for your dictionary
 let dict = dictf false // Uncomment if using a trie for your dictionary
 ```

Use the first one if you have implemented a Gaddag in `Dictionary.fs` and the second one if you have implemented a Trie.

- In `Program.fs`, in the `ScrabbleTemplate` project, in the `main` function, comment out the line that says:

```F#
let players = spawnMultiples "OxyphenButazone" Oxyphenbutazone.Scrabble.startGame 2
```

and uncomment the line that says:

```F#
let players    = [("Your name here", YourClientName.Scrabble.startGame)]
```

Change `"Your name here"` to the name of your Scrabble Bot (this is for pretty-printing purposes only) and change `YourClientName` to `ProjectId` from your `ScrabbleBot.fsproj` file.

Build and run your project. You should see something like this:

```
# # # # # # # # # # # # # # # # # # # #
# 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 space 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 be instantiated 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 servers internal game state is updated, but we have not updated the local client 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.

Writing a Scrabble bot

We have covered how to set up your project so that you can either watch Oxyphenbutazone play, or enter your own moves by hand.

To write a working Scrabble bot you must:

1. **Maintain a consistent state with the server** This includes keeping track of things like what pieces you have on hand, who's turn it is, how many players are left in

the game, what positions on the board are available to start new words on, etc.
2. **Finding valid moves to play** Find valid moves using the pieces you have on hand and the state of the board find. You may use whatever heuristics you like for this but a few examples could be

First move you find
Longest word you find
Highest scoring word you find
...

You are not required to calculate points to pass this assignment, but you are required to be able to stay in a game (continuously passing does not count)

3. **Communicate your moves to the server** You are given a robust protocol with which you can communicate with the server. We will cover these protocols in detail later, but they boil down to:

Send your moves to the server
Receive information about the success of your actions, as well as the actions of the other players, and update your internal state accordingly.

Aside from the server communication this part of the project is deliberately left open ended. How you maintain your state, and how you find which moves to play is up to you. A few words of caution though:

Warning: It may be tempting to create a function that given the tiles you have on hand returns all of the possible words you could write with them. This is a bad approach as it does not take into consideration the tiles that have already been placed on the board. In fact, this tactic only works for the very first word you place on the board and even then it does not work if you want to optimise for score. A much better approach is to start from a square on the board and build a word by incrementally interleave placing pieces from your hand and using ones that are already placed on the board. The `step` function from `Dictionary.fs` from Assignment 4 is useful here.

The ScrabbleUtil library

This library

- Provides a small set of datatypes for board representations (that you used to have in Assignment 7)
- Provides a wrapper for your dictionary so that it can be shared with other players and the server
- Contains functions for communicating with the Scrabble Server
- Contains several test boards that you can play off of.

Types

Coordinate systems 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.

Squares Squares have the following type

```
type squareProg = Map<int, string>
```

The string value in the map is a program written in our DSL (you can find several examples in Assignment 7) and the integer key is the priority of the function. This priroity is important when we calculating points and we will come back to this in detail later.

Tiles tiles are the pieces your place on the individual squares of the board. They have the following type

```
type tile = Set<char * int>
```

A tile can is a set of letters and their point values. For instance, the letter `A` which is worth one point is represented as the singleton set `{('A', 1)}`, while the wild-card tiles that can represent any letter are represented as a set of pairs of all letters in the alphabet each worth zero points.

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 word must be placed.

There is also a `usedSquare` that we use to calculate points for tiles 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 squares.

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

From Assignment 7 you already have the code required to parse values of type `boardProg` into a usable data structure.

The Dictionary wrapper

Building dictionaries takes time, especially if you have implemented a GADDAG, and having all clients and the server build their own dictionaries at startup takes even longer. We therefor provide a wrapper that allows you to pass your own dictionary to the server and to other clients (and your own client).

The library `ScrabbleUtil` contains the following module.

```
module Dictionary =
    type Dict

    type 'a dictAPI =
        (unit -> 'a) * // empty
         (string -> 'a -> 'a) * // insert
         (char -> 'a -> (bool * 'a) option) *   // step
         ('a -> (bool * 'a) option) option // reverse

    val mkDict<'a> :
        string seq -> // word list (your dictionary)
        ('a dictAPI option) ->
        (bool -> Dict)

    val test : string seq -> int -> Dict -> string list

    val isGaddag : Dict -> bool
    val lookup : string -> Dict -> bool
    val step : char -> Dict -> (bool * Dict) option
    val reverse : Dict -> (bool * Dict) option
```

The module contains a type `Dict` which is the dictionary that all bots and the server use. It provides three functions that these can use:

* `lookup : string -> Dict -> bool` that given a string `s` and a dictionary `d` returns `true` if `s` is in `d` and false otherwise.

- `step : char -> Dict -> (bool * Dict)` that given a character `c` and a dictionary `d` takes one step down the trie and returns a tuple `(b, d')` where `b` is `true` if traversing `c` completed a word and `false` otherwise, and where `d'` is the next level of the trie.
- `reverse : Dict -> (bool * Dict)` behaves the same way as `step` but parses the special reverse character. Only works on GADDAGs.

There is also the `isGaddag` function but this is only really useful if you want to create a bot that can play using both types of dictionaries. Oxyphenbutazone can do this, but there is no reason for you to support both.

There is also a `test` function that you can use to test that your dictionary works. It takes a sequence of strings (that you used to build your dictionary and which it checks for membership), an integer which determines how many incorrect matches it will return (default is set to 10) and your dictionary. If this function returns an empty list that means that your dictionary works as well as the one that Oxyphenbutazone uses. For the final tournament you must pass this test or use the cut-down dictionary that only handles words of odd length.

To construct the dictionary you need to parametrise it with the dictionary functions you created in Assignment 4.

The function `mkFun` takes functions `empty`, `insert`, and `step` from your Assignment 4, an optional argument for the `reverse` funtion if you have implemented a GADDAG rather than a trie, a sequence of words to place in the dictionary, and returns a function of type `bool -> dict` that you will have to instantiate with `true` if your bot expects a GADDAG, and `false` if it expects a trie. For an example of how this function is used look at `Program.fs` in the `ScrabbleTemplate` executable project.

There is an important restriction to this setup.

- If `Dict` is instantiated with a GADDAG (a reverse funtion is provided) then it can be used by bots that ***either*** use a GADDAG or a trie.
- If `Dict` is instantiated with a Trie (a reverse function is not provided) then it can ***only*** be used by bots that use a trie (not a GADDAG).

What this means in practice is that if you have implemented a GADDAG then you can import all bots, even those that use tries and give them your dictionary, but if you have implemented a trie you cannot give that to bots that expect a GADDAG.

Important: The fact that a trie cannot be used as a dictionary for a bot that expects a GADDAG is not counted negatively in any way. A GADDAG is an optimisation for those who want faster bots, they are not strictly necessary.

Oxyphenbutazone works on both tries and GADDAGs.

The ScrabbleLib library

This library contains a simple board parser that you can use if you have not finished Assignmens 6 and 7.

```
parseSimpleBoardProg : boardProg -> coord -> bool
```

That given a board program `bp` returns a function that takes a coordinate `c` and returns `true` if `c` is on the board and `false` if it is outside. It does not say anythin about the actual square that `c` is on and cannot be used to calculate points. It can, however, be used to play on more advanced boards which is required to get full poinst on the assignment if you are aiming for that.

Important!!!

This function should only be called once as parsing the board takes time. Call it before the game starts with the board program and then use the resulting function of type `coord -> bool` to analyse specific coordinates. Do not re-parse the board multiple times.

Setting up the bot

Your scrabble client exposes a single function that the server can call to set up a game.

```
val startGame :
boardProg ->                  (* Scrabble board *)
(bool -> Dictionary.Dict) -> (* Dictionary (call with true if using a
GADDAG,
and false if using a Trie) *)
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 setup *)
```

This function takes everything you need to set up your game. A `boardProg` that you can parse using your parser from Assignment 7, a dictionary as describe in the `ScrabbleUtil` section (initialise the function with `true` if your bot uses a GADDAG,

and false otherwise), the number of players, your player number (determined by the server), the starting player's number, your starting hand, a tile lookup table, an optional timeout for the maximum amount of time your client has to submit a move (only needed if you want six points on the assignment), and a stream that we use to communicate with the server.

If you take a closer look at the tile lookup table you can see that it is a map from `uint32` to `tile` (tiles are described in the `ScrabbleUtil` section). This map is provided at startup but the server will only from this point onwards communicate tiles by their identity number. An example of this can be seen in the starting hand which has the type `(uint32 * uint32) list` where the first element of each pair in the list is the identifier for a tile (a key in the lookup table) and the second is the amount of those tiles that you are given. We will later cover the protocol that communicates with the server which contain these identifiers rather than the tiles themselves.

Setting up the server

The server is set up for you. It exposes a single setup function. All other communication between the server and the clients is handled over the network. The server is instantiated in the `Program.fs` file in the `ScrabbleTemplate` project via a `startGame` function.

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

The first arguments (Board-random seed) are the same as for your client setup. The port determines which port you communicate on (you should never have to change this, but sometimes it can help to be able to switch the port). Finally there is a list of pairs of client names (for pretty-printing purposes) and functions that have the same type as the clients only exposed `startGame` function. To see an example of how

clients are communicated with the server have a look at the `ScrabbleTemplate` project and the `Program.fs` file.

Maintaining a 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. In `Scrable.fs` in the `ScrabbleBot` project you will find the following data structure.

```
type state = {
        board         : Parser.board
        dict          : ScrabbleUtil.Dictionary.Dict
        playerNumber  : uint32
        hand          : MultiSet.MultiSet<uint32>
        }
```

The state record contains four fields `board`, `dict`, `playerNumber` and `hand`, where `board` is the board you are playing on (parsed by the functions you wrote in Assignment 7), `dict` is the dictionary you are using (wrapped by the `ScrabbleUtil` library), `playerNumber` is your player id, and the field `hand` contains the current tiles you have on hand as a multiset of integers. In combination with the tile lookup table 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, how many players there are, who has forfited the game, etc. 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.

The main game loop

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 that communicates using a stream called `cstream`.

This is a high-level description. We will cover the exact protocol used to communicate with the server in the next section.

```
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 space 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 require that you use these rather than `printf` as they can be switched off easily when they are no longer needed.

Look over this loop and try to see its internal logic and how it works. At the moment, it sends moves to the server that you type on the keyboard -- `let move =`

`RegEx.parseMove input`.

The main goal of this project is to replace this line with a call to a function that finds `move` for you.

Server communication

The following type detail the information that is sent to and received from the server. They are covered in detail below but this gives a good overview.

```
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 types of messages you can send to a server 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 list of gameplay errors is returned to the player who made the move
- 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:	`GPEEmptyMove`
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:	`GPEFirstWordTooShort`
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.

Calculating points

Calculating points is not necessary to pass the project, but it is necessary to play well.

What makes point calculation interesting is that there are squares on the board that take the entire word you place into consideration not just the tile you place on top of them.

Recall that we have the following tyes:

```
type word = (char * int) list
type squareProg = Map<int, string>
type square = Map<int, word -> int -> int -> int>
val : parseSquareProg : squareProg -> square
```

Every square can have `SquareProg` (unless it is outside the board) that details how points are calculated. The key of the map is the priority of the function -- the lower the number the sooner that function will be run. The general idea is that after having placed tiles over a set of squares, all functions are collected, sorted according to their priority and run in order passing the results from one into the next. Assignment 2 makes this precise. The `string` value is a program itself and it is written in our DSL. The program has two special variables `_pos_` and `_acc_` where `_acc_` is the number of points that have been calculated and `_pos_,` which is the position in the word of the character that is placed over this square. Finally the variable `_result_` stores the return value of the program that is passed on to the next function in the priority chain.

For a standard Scrabble board, the program looks like this (as described in Assignment 7).

```
let singleLetterScore =
        Map.add 0 "_result_ := pointValue(_pos_) + _acc_" Map.empty

let doubleLetterScore =
        Map.add 0 "_result_ := pointValue(_pos_) * 2 + _acc_" Map.empty

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

```
let doubleWordScore = Map.add 1 "_result_ := _acc_ * 2" singleLetterScore
let tripleWordScore = Map.add 1 "_result_ := _acc_ * 3" singleLetterScore
```

After compiling these functions with `parseSquareProg` you will get actual `F#` functions in the `square` type that have been constructed by parsing and evaluating programs in `squareProg`.

We will see how this works through an example:

We will use SLS to mean single letter score, DWS to mean double word score, and so on.

Consider that we are placing the word QIN (`[('Q', 10); ('I', 1); ('N', 1)]`) over the tiles triple letter score, single letter score, and double word score. By following the rules of Scrabble this should calculate to `64` points. We have three squares of the form

1. `map [(0, TLS)]`
2. `map [(0, SLS)]`
3. `map [(0, SLS); (1, DWS)]`

Where SLS, TLS, and DWS have the type `word -> int -> int -> int`

By partially applying these function with the word QIN and the position of the letter that is placed on the tile, we get the following maps (your evaluation function from Assignment 6 and seven will already do this)

1. `map [(0, fun acc -> 10 * 3 + acc)]`
2. `map [(0, fun acc -> 1 + acc)]`
3. `map [(0, fun acc -> 1 + acc); (1, fun acc -> acc * 2)]`

By collapsing the map into a list, sortying by priority, and keeping the functions, we get the following functions in order.

1. `fun acc -> 10 * 3 + acc`
2. `fun acc -> 1 + acc`
3. `fun acc -> 1 + acc`
4. `fun acc -> acc * 2`

Composing these with function composition or pipes and instantiating the accumulator to `0` gives us the following evaluation.

```
0 |> fun acc -> 10 * 3 + acc |> fun acc -> 1 + acc |>
fun acc -> 1 + acc |> fun acc -> acc * 2 ==>

10 * 3 + 0 |> fun acc -> 1 + acc |>
fun acc -> 1 + acc |> fun acc -> acc * 2 ==>

30 |> fun acc -> 1 + acc |>
fun acc -> 1 + acc |> fun acc -> acc * 2 ==>

1 + 30 |> fun acc -> 1 + acc |> fun acc -> acc * 2 ==>

31 |> fun acc -> 1 + acc |> fun acc -> acc * 2 ==>

1 + 31 |> fun acc -> acc * 2 ==>

32 |> fun acc -> acc * 2 ==>

32 * 2 ==>

64
```

Which is indeed the expected result.

Your solution for Assignment 3.8 (which uses your solution from Assignment 2.17) already does the lion's share of the work. but the square type in 3.8 was `type square = (int * squareFun) list` rather than a map. Converting your map to a list and appealing to 3.8 calculates the points.

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).