IT UNIVERSITY OF COPENHAGEN

Functional Programming

Jesper Bengtson

The scrabble project

What do I need to pass?

- Hand in your project by the 7:th of May at 08:00
- Worth six points in total
  - 2 points - finish a game against yourself on an infinite board (you do not have to count points nor play well)
  - 1 point - Multiplayer and implement your dictionary
  - 1 point - Use the DSL to play on all boards. You will be given a simple parser or use your own.
  - 1 point - Parellise your algorithm
  - 1 point - Respect the timeout flag

What do I need to pass?

- Hand in your project by the 7:th of May at 08:00
- Worth six points in total
  - 2 points - finish a game against yourself on an infinite
  - ll)
  - be
  - 1 point - Parellise your algorithm
  - 1 point - Respect the timeout flag

You have a few days more than a month. Start immediately. The exercise sessions are manned throughout the project.

Make use of them.

What we give to you

- The Scrabble server
- A dictionary (not a good one, but a fast one)
- A parser (only tells you if you are inside a board)
- A code skeleton that includes communicating with the server. You will have to handle sending and receiving messages but not setting up the infrastructure that allows you to do this.
- Jesper's bot Oxyphenbutazone to play against

What we do not give you

This is an open-ended project

- A means to parse the Scrabble boards
- A fully functioning dictionary
- A complete board parser
- Means to count points
- Heuristics to play Scrabble

Communicating with the server

The server will

- Run several scrabble games concurrently
- Allow for creating games with an arbitrary (within reason) number of bots
- Accept input from the players in order
- Evaluate the move from the player
- Return acceptance or error messages
- Broadcast results of each players actions to the other players (but not the error messages)

Communicating with the server

The player must

- Calculate their moves
- Send these moves to the server
- Accept feedback from the server
- Maintain a consistent state with the server on what letters have been placed where (how many points you have is not critical but useful - the server knows)
- Provide feedback in reasonable times. Timeouts will be enforced (for one point, not mandatory)

Prerequisites

- You must be able to handle .NET solutions through an IDE of your choice
- .NET 7.0

Dependencies

- You will get a template project with all dependencies provided as dlls packages
  - ScrabbleServer (the scrabble server)
  - ScrabbleUtil (a core library with useful functions)
  - ScrabbleLib (an auxiliary library containing a scaled-down parser)
  - Oxyphenbutazone (Jesper's bot that invariably gets beaten every year)

Two points (mandatory)

- Finish a game against yourself on an infinite board
  - You do not have to play well
  - You do not have to calculate points
  - You do not need to parse boards or their tiles
  - You do not have to implement a dictionary
  - You do have to keep track of what letters have been placed on which squares and the letters you have on hand

Let's have a look

Plan of attack for the project

- Get familiar with the main game loop and how your client communicates with he server
- Write a function that given a set of pieces, a board, a coordinate on the board, and a dictionary calculates a valid word to place on the board
  - From the start take pieces on the board into account

Plan of attack for the project

- Get familiar with the main game loop and how your
- a
  - a
  - From the start take pieces on the board into account

Warning!!!

Do not try to calculate moves based only on the tiles you have on hand. This will only work for the very first move if you do not calculate points and never if you do.

The game state

The game state needs to keep track of

- Tiles on hand
- Tiles placed on the board
- Possible starting coordinates
- … many other things

Dictionary and multiplayer (one point)

To get another point (or just have fun) write a Trie for your dictionary and allow multiplayer

This is low hanging fruit. A trie implementation is around thirty lines of code

Program state for multiplayer

- The state should keep track of
  - Who am I?
  - Who's turn is it?
  - Has anyone forfeited the game or been kicked out (important in order to calculate the next player)
- The state should be updated whenever you or anyone else places something on the board

The dictionary

We provide a mean to instantiate a dictionary that we can then send to multiple bots

- Having every player compute a dictionary took too long in tournaments
- This allows us to provide a default dictionary to those who have not implemented a Trie
- Allows people who use GADDAGs to play with those who use Tries (but not the other way around, and this is fine)

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
         (unit -> 'a) *
         (string -> 'a -> 'a) *
         (char -> 'a -> (bool * 'a) option) *
         ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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 dictionary module

```
module Dictionary =

  type Dict

  type 'a dictAPI =
       (unit -> 'a) *
       (string -> 'a -> 'a) *
       (char -> 'a -> (bool * 'a) option) *
       ('a -> (bool * 'a) option) option

  val mkDict<'a> :
      string seq ->
      ('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 type of the Dictionary that all bots use

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

If you have not implemented a dictionary, the API should be None

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

If you have created a dictionary, you give it

A function to construct the empty dictionary

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

If you have created a dictionary, you give it

an insertion function

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

If you have created a dictionary, you give it

a step function

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

If you have created a GADDAG provide

a reverse function (and None if you have a Trie)

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

Given a sequence of words (all words in the English language for instance, and an optional dictionary API mkDict returns a dictionary that you can use

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

Call this function with true if your bot plays using a GADDAG and false otherwise. This lets Trie bots play using GADDAG dictionaries even though they do not make full use of them

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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 test function can be used to test that your Trie or Gaddag is correct (the integer argument is the maximum number of false hits you can report back)

The dictionary module

```
module Dictionary =

    type Dict

    type 'a dictAPI =
          (unit -> 'a) *
          (string -> 'a -> 'a) *
          (char -> 'a -> (bool * 'a) option) *
          ('a -> (bool * 'a) option) option

    val mkDict<'a> :
        string seq ->
        ('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
```

When your bot is playing it only needs these two functions (only the step function if you are using a Trie)

Parsing the board and counting points

To get another point you must be able to
Play and finish games on all boards

Program state

Similar as before, but additionally

- Your anchor points (the places where you can start to write words) need to keep track of all possible starting points (this was less important on an infinite board since you cannot get stuck)
- You must be able to start writing before words that have already been played or you will run out of space.
- It may be necessary to keep track of tiles and their values to compute score.

Let's have a look

Parallelism

You must parallelise your algorithm

- Search for words in parallel
- Different paths of the word in parallel (be careful here)

Respect the timeout flag

You must respect the timeout flag for another point

- Abort search ahead of time and play the best move you have found.
- Be careful not to get out of synch (end up in a state where the server has another view than you do)
- We will not give insanely short timeouts, but you should be able to handle a second or so.

Questions?