0 Obtaining the code

The code for the assignment can be downloaded from here .

See the instructions in Weekly Exercises for how to unpack the code

1 Deadline and submission

The deadline for this assignment is Monday the 15th of July, 18:00.

Late submissions are accepted up to five days after the deadline, but at a penalty: 5% off your total mark per day. Submissions more than 5 days (120 hours) after the deadline are not accepted. To ward against network issues, there will be a 1 hour grace period where no penalty will apply.

We cannot issue individual deadline extensions unless:

- You have an ELP , or
- you apply for, and you're granted, special consideration .

While connected/using a CSE machine, type the following to submit your work from the directory within the assignment folder:

```
$ give cs3141 assign1 MoveGenerator.hs
```

Alternatively, you can use the give web interface or submit through webCMS.

Your submission should work on the ghc version on the CSE lab system, which is ghc 8.8.4 . When you submit, some basic tests will be run. These tests are *not* sufficient, and you should additionally run your own tests.

Do not remove any of the declarations from the template, do not change their type, and do not move them into a local scope (such as a let or where block). We won't be able to test your submission if you do.

Note that you will submit *only* the Haskell module file called MoveGenerator.hs (and *not* the entire project).

2 Overview

In this assignment, you will apply the skills you have developed in the course so far to two interlocking tasks: developing parts of a trie library, and developing a move generator for crossword games. The most popular such game is Scrabble , but what we'll do here is general enough to also apply to ~~ripoffs~~ other similar games such as Words With Friends, WordFeud etc.

3 Task 1: A Trie Library (8 marks)

A dictionary, for our purposes, is a set of words. A word, for our purposes is a String .

A Trie is a data structure for representing dictionaries. Its main advantage is that many operations, such as lookup and insert, have runtime that's proportional to the length of the word, not the size of the dictionary.

```
data Trie = Trie Bool [(Char,Trie)] deriving (Eq,Show)
```

A node Trie b ts consists of:

1. b :: Bool indicating whether the empty word is included in the dictionary or not.
2. A list ts :: [(Char , Trie)] of pairs (x,t) of characters x and associatied subtries t . The intention is that x:xs is in the dictionary Trie b ts whenever xs is in the dictionary t . In a well-formed Trie, these lists must be sorted by character, and cannot contain duplicate entries for any character.

For example, the dictionary ["he" , "hell" , "hello" , "hi"] would be represented by the following Trie :

$$\begin{tikzcd}
 & F \arrow[d, "h"'] & \\
 & F \arrow[dl, "i"'] \arrow[dr, "e"] & \\
T & & T \arrow[d, "l"] \\
 & & F \arrow[d, "l"] \\
 & & T \arrow[d, "o"] \\
 & & T
\end{tikzcd}$$

...which looks like this in our Haskell representation:

```
Trie False
 [('h',
  Trie False
   [('e',
    Trie True
     [('l',
      Trie False
```

```
        [('l',
         Trie True
          [('o',
            Trie True [])
          ]
         )]
     )]
  ),
  ('i',
   Trie True [])])]
```

This part of the assignment consists of the following subtasks:

1. Implement fromList :: [String] -> Trie .
2. Implement wellFormed :: Trie -> Bool .
3. Implement minimal :: Trie -> Bool .
4. Implement prune :: Trie -> Trie .
5. Implement check :: Trie -> String -> Bool .
6. Implement union :: Trie -> Trie -> Trie .
7. Implement intersection :: Trie -> Trie -> Trie .
8. Define an appropriate Monoid instance for tries.

Correct implementations of the above are worth one mark each.

Note that as the author of the Trie library, **you are responsible for maintaining well-formedness** , and this is always an implicit correctness criterion. In other words, all functions with codomain Trie should produce a well-formed Trie , assuming their input Trie s (if any) are also well-formed. You are *not* required to maintain minimality, although it's still suggested that you do.

More detailed instructions, including descriptions of what exactly these functions should do, can be found in the code template.

4 Task 2: A move generator (12 marks)

4.1 Introduction

In the remainder of the assignment, we will use our Trie library above to develop a move generator for Scrabble-like word games.

A player makes a *move* by placing a sequence of tiles on the board, either horizontally or vertically. These tiles will connect with the pre-existing letters on the board to form words. A move is *legal* if every word of length >= 2 thus formed occurs in the dictionary, and if at least one word of length >= 2 is formed in the direction of play. A word counts as *formed* if it includes at least one newly placed tile.

A move generator is a core component of any word game program. It takes the following inputs:

1. A Trie , representing the dictionary
2. An Int , representing the number of tiles the player will place.
3. A Rack . This represents the player's pool of letter tiles available for play.
4. A Board . This represents the current state of the board.

Its output is a Trie , representing the subset of the dictionary that are legal moves that can be played using exactly the desired number of tiles from the rack.

We make a number of simplifying assumptions:

1. We will only generate moves that start from a particular fixed square on the board.
2. We only consider horizontal moves. Therefore, a legal move must create a word of at least length 2 horizontally.

4.2 Data types

The move generator introduces a number of new data types.

4.2.1 Constraints

Our first type, the Constraint , is used to do (a simplified version of) regular expression matching on words:

data Constraint = Wildcard | Mem String deriving (Show,Eq)

A Constraint represents a predicate on characters. A character c is said to **match** a constraint according to the following clauses:

1. Any character c matches Wildcard .
2. A character c matches Mem cs , if c occurs in cs .

4.2.2 Patterns

A Pattern is a list of Constraint s:

type Pattern = [Constraint]

We say that a word xs matches a pattern cs if:

1. length xs == length cs , and
2. the i:th character of xs matches the i:th constraint in cs for all i.

4.2.3 Tiles

A Tile is either a letter tile, or a blank tile. Blank tiles are the most OP thing in the game: they are wildcards that can be played as if they were any letter.

data Tile = Letter Char | Blank deriving (Eq,Show)

A Rack is just a list of Tile s.

type Rack = [Tile]

4.2.4 Boards

The following type models the game board:

newtype Board = Board [((String,String),Maybe Char)] deriving (Eq,Show)

This type is not meant to model the whole game state, but to contain only those aspects which are relevant for making our move.

The Board consists of a list of columns from left to right, where each column is represented by a triple ((above,below),on) representing the tiles immediately above the main row, immediately below the main row, and on the main row.

More details are available in the Haskell template. Detailed real examples can be found further down.

4.3 The task

The ultimate task here is to implement the move generator. This task is subdivided into several auxiliary functions:

1. Implement pick :: Eq a => [a] -> a -> (Bool ,[a]) (1 mark)
2. Implement sandwichableLetters :: Trie -> String -> String -> [Char] (1 mark)
3. Implement filterLength :: Int -> Trie -> Trie (1 mark)
4. Implement filterPattern :: Pattern -> Trie -> Trie (2 marks)
5. Implement filterPlayables :: Rack -> Trie -> Trie (2 marks)
6. Implement moves :: Trie -> Int -> Rack -> Board -> Trie (4 marks)
7. Implement allMoves :: MoveGenerator -> Trie -> Rack -> Board -> Trie (1 marks)

More detailed specifications of each function can be found in the Haskell template.

Note that **you are still responsible for maintaining well-formedness** .

The big one here is moves . Everything that comes before it in this task (and most things from Task 1) are meant to be suggestive of an algorithm for the moves function. Still, finding this algorithm may require some non-trivial thinking; do allow time for that.

5 Move Generator Examples

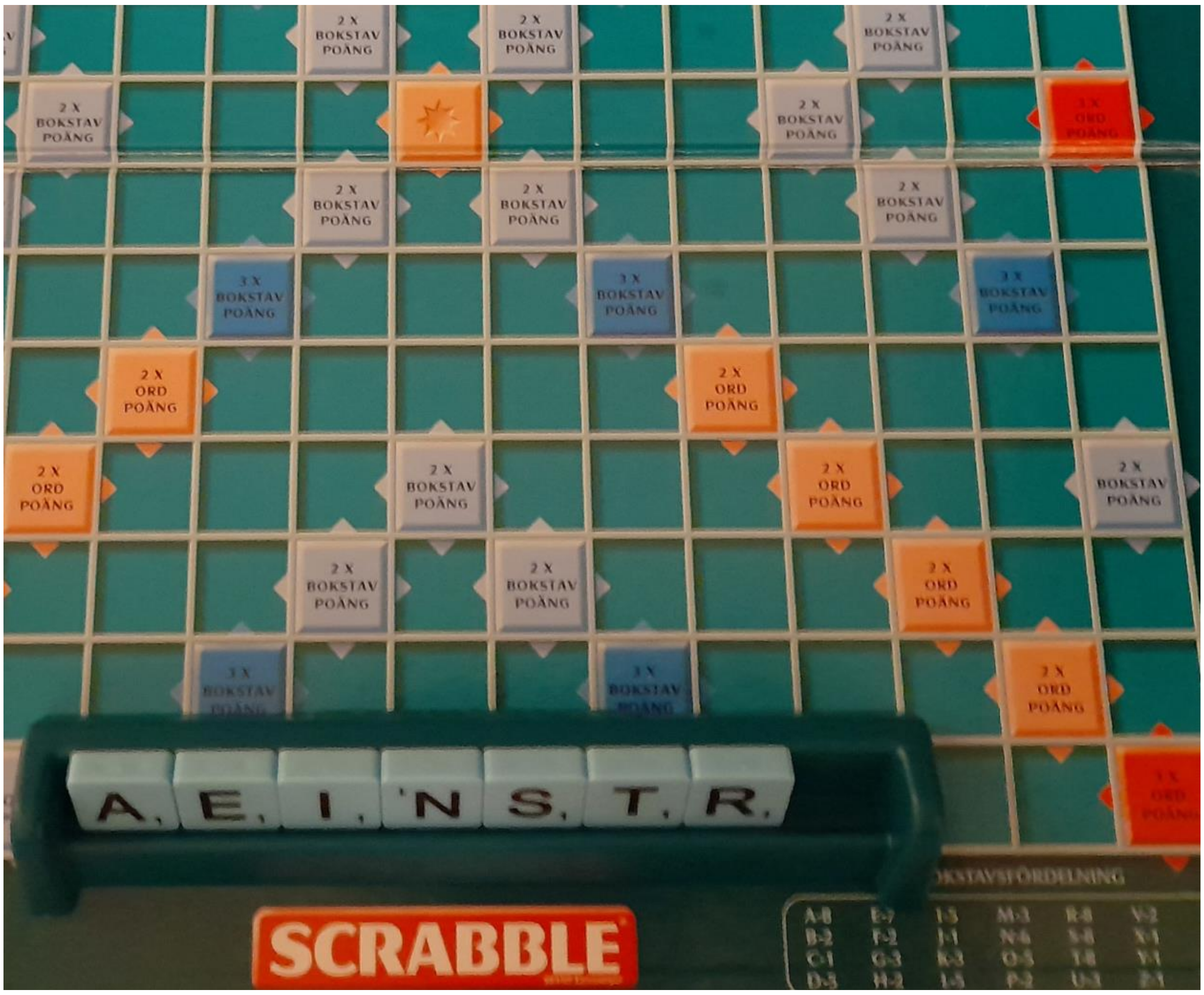
These examples are intended to build an intuition for what the move generator should do. In these examples, assume the following conventions:

1. The pink-ish square with a star in it is the *origin point* , where we assume the first tile of our move will be placed.
2. White tiles represent letters that were already on the board before our move.

3. Blue tiles represent the tiles we place on the board as part of our move, or the tiles we have at our disposal on the rack.

5.1 With an empty board

In this position, the player has the letters AEINSTR on their rack. The board is empty: no tiles have previously been played. Starting from (and including) the origin point, there are eight columns before the edge of the board.



This rack and board state would be represented as follows in our Haskell encoding:

```
empty_board :: Board

empty_board =

 Board

  [(("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),
   (("",""),Nothing),
   (("",""),Nothing),
   (("",""),Nothing)
  ]


aeintsr_rack :: Rack
aeintsr_rack =
 [Letter 'a',
  Letter 'e',
  Letter 'i',
  Letter 'n',
  Letter 's',
  Letter 't',
  Letter 'r'
 ]
```

Here's an example of a legal move in this position (for 66 points, assuming the standard Collins dictionary):

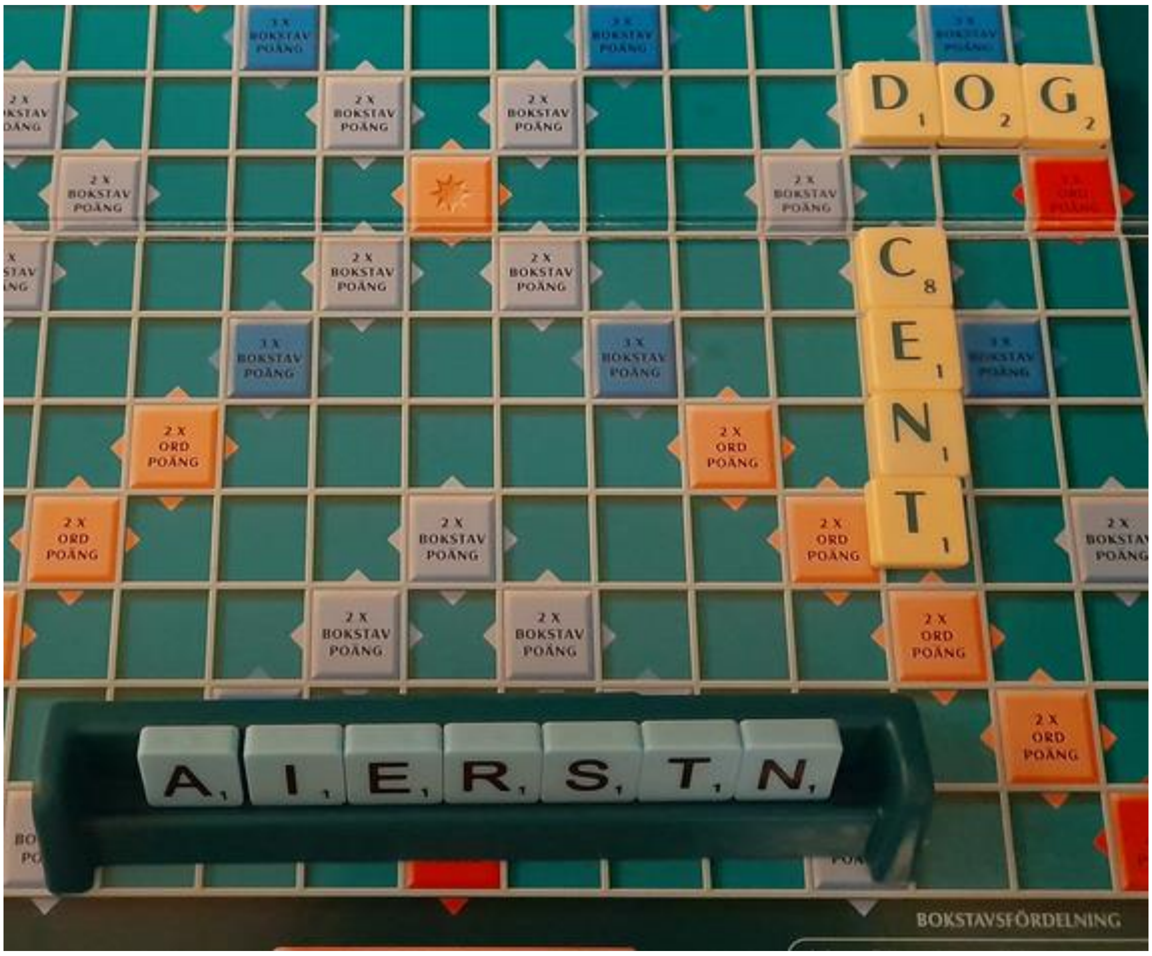
Therefore we would expect that, if the dictionary t :: Trie contains the word "nastier" , then the following should be True :

check (moves t 7 aeintsr_rack empty_board) "nastier"

5.2 Vertical Words

Now let's introduce a complication into the example: pre-existing tiles that will form words vertically as we touch them.

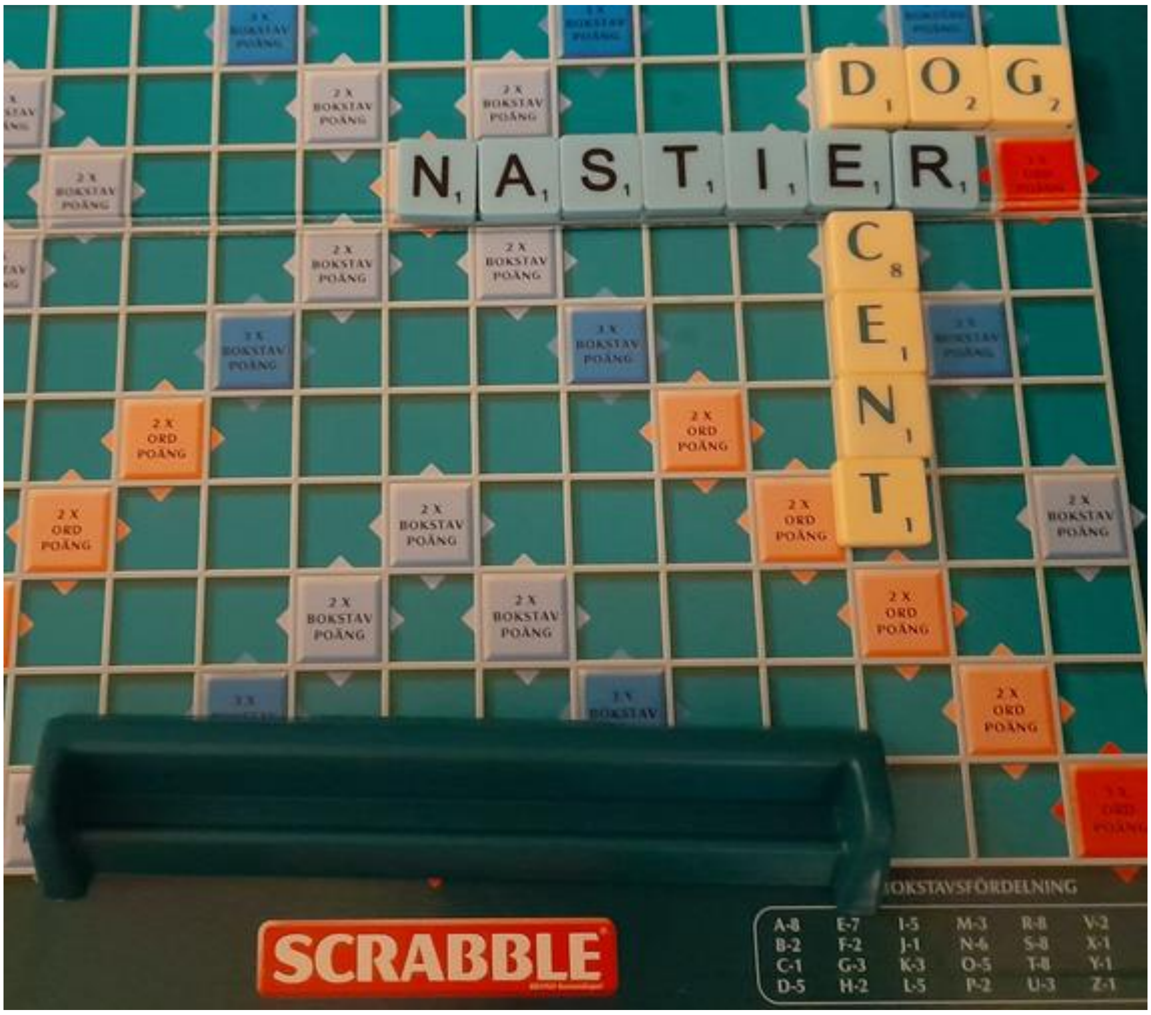
In this position, the player again has the letters AEINSTR on their rack. But the board is not empty. Instead, there are letters above and below the main row that we must take into account.



This board state would be represented as follows in our Haskell encoding:

```
dog_board :: Board

dog_board =

 Board

  [(("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),

   (("",""),Nothing),

   (("d","cent"),Nothing),

   (("o",""),Nothing),

   (("g",""),Nothing)

  ]
```

NASTIER is still a legal move (assuming the standard Collins dictionary):



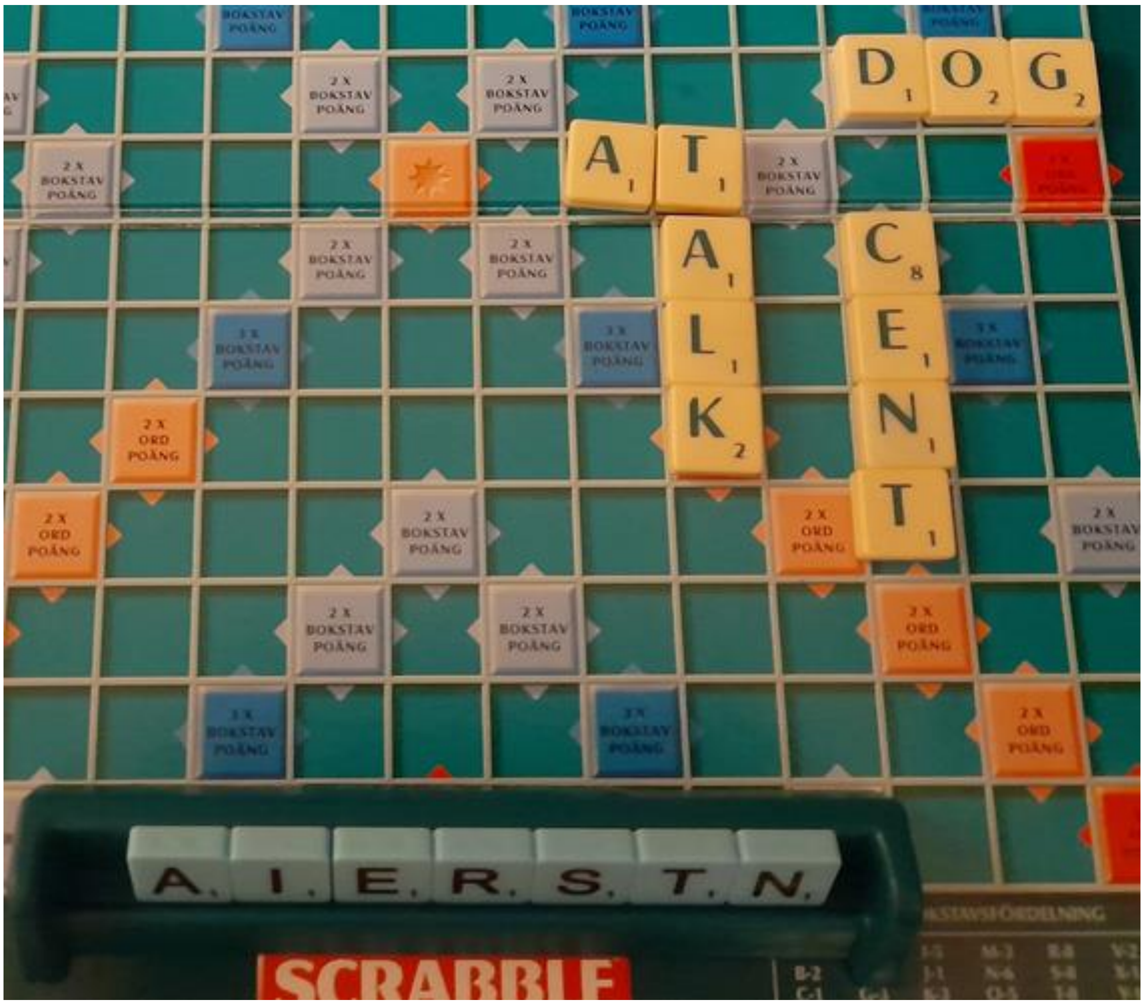
But note that this play now forms three words: NASTIER, DECENT, and OR. In order for this move to count as legal, all three need to be in the dictionary.

Therefore we would expect that, if the dictionary t :: Trie contains the words "nastier" , "decent" , and ~~"dog"~~"or" , then the following should be True :

check (moves t 7 aeintsr_rack dog_board) "nastier"

5.3 Tiles on the main row

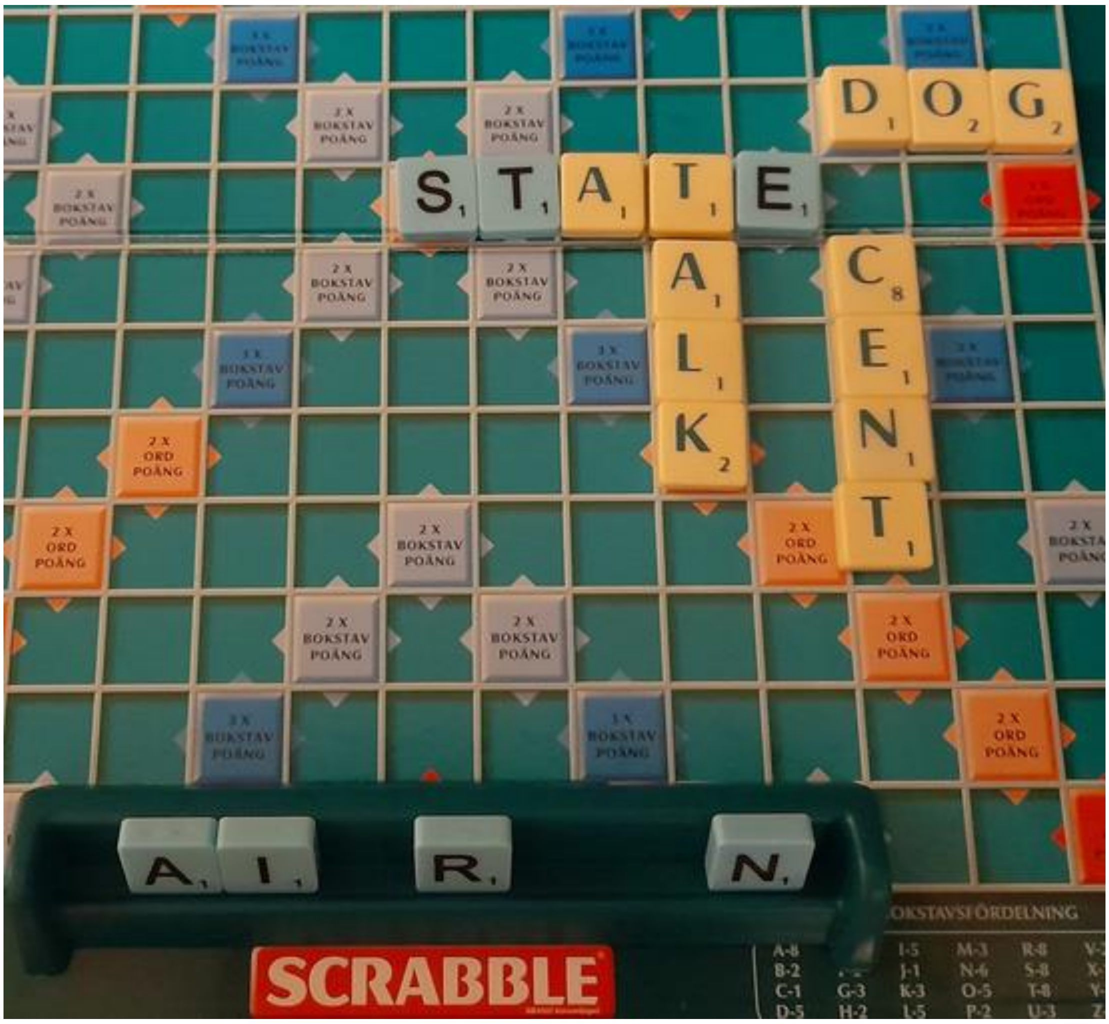
For our next complication, what happens if there are already tiles present in the main row? The player still has the letters AEINSTR on their rack.



This board state would be represented as follows in our Haskell encoding:

```
at_board :: Board
at_board =
  Board
   [(("",""),Nothing),
    (("",""),Nothing),
    (("",""),Just 'a'),
    (("","alk"),Just 't'),
    (("",""),Nothing),
    (("d","cent"),Nothing),
    (("o",""),Nothing),
    (("g",""),Nothing)
   ]
```

STATE is a legal move here (assuming the standard Collins dictionary):



This move places the tiles STE to form the word STATE around AT. Note that this move is legal whether or not TALK is in the dictionary: even though TALK shares a letter tile with STATE, it was already there before our move, and only newly formed words should be checked.

Therefore we would expect that, if the dictionary t :: Trie contains the word "state" , then the following should be True :

check (moves t 3 aeintsr_rack at_board) "state"

6 Hints

6.1 Test, test, test!

You can and should write your own tests to convince yourself that your code is correct. For your QuickChecking convenience, Arbitrary instances have been supplied for all types except Board .

It's not completely obvious how to define a useful generator for Board . Putting out totally random characters is unlikely to form word stems, and would just block. It would probably have to be parameterised on the dictionary to be of much use; if you come up with one, feel free to share on the forums.

6.2 Obtaining a dictionary

If you want to try out your move generator on a real dictionary, the canonical choice is Collins Scrabble Words (CSW). For legal reasons I won't redistribute it here, but you can get a list of all the words (without definitions) as follows:

1. Download the Linux version of Zyzzyva .
2. Poke around in the tarball for e.g. CSW19.txt .
3. Copy it to your working directory, and run the following in GHCi:

```
csw <- dictionaryFromFile "CSW19.txt"
```

This assumes you've implemented fromList . Then you can play around with the massive csw :: Trie and see how fast your implementation chokes. If this seems surprisingly fast, remember that Haskell is lazy and will defer construction of the trie until there's demand for it.

Zyzzyva is the official word study tool of the north American Scrabble players association, and as such they distribute this word list with permission. Unless you too have a licence from HarperCollins, you should probably stick to personal use.

You can also use Zyzzyva as a reference implementation for some of the tasks. For example, it can tell you what the 10 anagrams of NASTIER are; on the empty board, a correct move generator should return all 11.

Don't worry if your move generator isn't performant enough to handle these. We won't stress-test them to that extent.

6.3 The Secret Escape Hatch

If you find tries very confusing, there is an escape hatch you can use. You're given a toList as part of the template. Once you have fromList , you can implement most Trie -processing function by converting to lists, doing the thing, and then converting back again.

This is a bit cheap, and will produce outrageously slow code. I would only do it as a last resort. But if you're desperate, it's an option, and should cost you no marks, except in the unlikely event that your code is so heroically slow that our test runners time out. If you resists this temptation though, there's a reward:

If you've consistently used tries instead of lists, and resisted the temptation of using this escape hatch, **and** if additionally your assignment scores 18/20 or better, then you are eligible for **two bonus points for the final exam** . In order to certify that you have refrained from using this escape hatch, set the flag trieOrDie :: Bool to True to certify that you have indeed used Tries consistently.

7 Asides

7.1 Legalese

Scrabble is a trademark of Hasbro and/or Mattel, depending on where in the world you are.

7.2 What next?

Once you have a move generator like the one above, it's relatively easy write a program that can beat most humans: just pick the highest-scoring of the moves you generated!

To give even world champions a run for their money, you would additionally want to incorporate the quality of your leftover tiles into your valuation, and add some lookahead.

8 The Fine Print

8.1 Marking and testing

All marks for this assignment are awarded based on *automatic marking scripts* . Marks are not awarded subjectively, and are allocated according to the correctness criteria outlined for each subtask. The scripts that run when you submit the assignment are similar to the scripts that will be used to determine your final marks. But they are not the same, and not sufficient. You are advised to do your own testing, instead of relying solely on the submission test suite.

Barring exceptional circumstances, the marks awarded by the automatic marking script are *final* . For this reason, plese make sure that your submission compiles and runs correctly on CSE machines, and that the submission scripts do not report any problems.

8.2 Plagiarism

All work submitted for assessment must be **entirely your own** . Unacknowledged copying of material, in whole or part, is a serious offence. Before submitting any work you should read and understand the UNSW Plagiarism Policy.

Submission of any work derived from that of another person, or solely or jointly written by or with someone else, without clear and explicit acknowledgement, is considered student misconduct and will be prosecuted accordingly; possible consequences include automatic failure of the assignment and an overall mark of zero for the course. This includes using unreferenced work taken from books, web sites, etc.

Do not share your work with any other person! Allowing another student to copy you work will, at the very least, result in zero for that assessment. If you knowingly provide or show your work on this assignment to another person for any reason,, and work derived from it is subsequently submitted, you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep all your work private. If you are unsure about whether certain activities constitute plagiarism, you should ask us before engaging in them!

8.3 What about generative AI?

Including unattributed code produced by generative AI such as ChatGPT is here treated as a special case of what is described above: work not entirely your own, and may be prosecuted accordingly.

This does not mean all use of generative AI is off the table. By all means go ask your favourite chatbot to teach you about Haskell, about property-based testing, or about mathematically structured software development, so long as it is in general terms and not directly applicable to the assignment. Asking your favourite chatbot to write a move generator, or a trie library, or parts thereof, is over the line. If you're unsure whether what you're doing is ok, you should ask before doing it.