Homework 10/11: Scrabble AI

Due: 3/31, 4/7

For these last 2 homeworks we're going to explore one of the classic uses of logic programming languages: making a game AI. Specifically, we're going to make a game AI to help us cheat at scrabble from homework 1. We will tackle this in 2 parts. The first part will just be coming up with a word given a list of tiles. The second part, we will put that word on the board. All of the work is going to be done inside the `ai.pl` file.

Homework 10: Trie some new patterns

For the first assignment we have 2 tasks. We want to be able to form words out of tiles, then we want to make a "pattern" out of those tile. We'll use this in the next homework when we're putting it on the board.

The first part is technical and uses a new data structure. Hang with it, there's not a lot of code to write once you get the idea. For the first part I have a group of tiles, I want to get every possible word I can make out of those tiles. For example, if I have the tiles `[p,a,n,t,s]` I can make the words. `pan, pants, pat, past, an, at, as, tap, span, san, sat, snap, stan`. We're going to do this with the `make_word` predicate

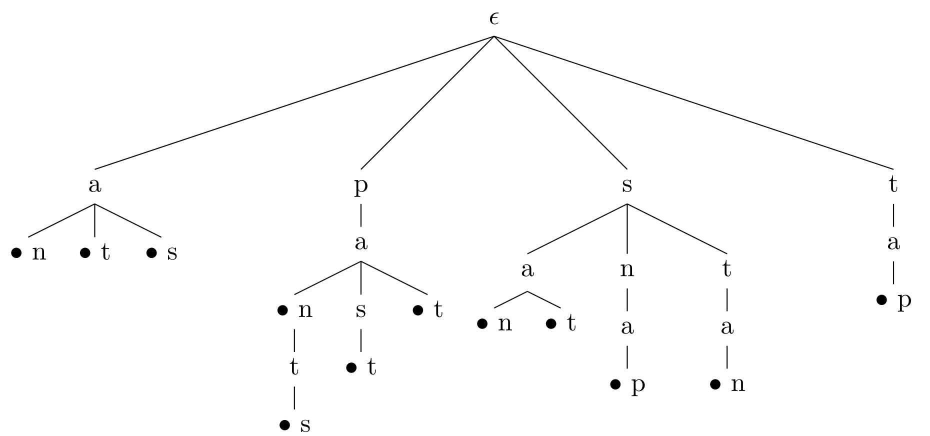
```
make_word(Tiles,Part,Word).
```

This predicate is true if, and only if,

- `Part` is a prefix of a word
- `Word` is a list of letters in the suffix of a word
- All letters `Word` are in `Tiles`.

This is a technical definition because of the strategy we're going to use to see if something is a word. One (inefficient) way of making a guess out of the tiles would be to permute the list of tiles, and seeing if they form a word. You can see in `trie.pl` that I have a predicate `word` that is true iff the argument is a word. Unfortunately, this is going to be too slow for us. We need something better.

Our solution is actually a pretty slick data structure called a trie. The goal of a trie is to encode a set of words, and make looking up words efficient. The idea is surprisingly simple. We make a tree, and put each word in the tree. The trick is that each node of the tree is the prefix of a word, and the branches represent the next letter. Lets see what this looks like with our list of words from before.
`pan, pants, pat, past, an, at, as, tap, span, san, sat, snap, stan`.



The bullets here tell us where we've found a word in the trie, and ϵ is our root node. It represents an empty string. In order to check if we have a word we just walk down the path corresponding to each letter. If we get to a word node, then we have a word, if we don't, or we run out of trie, then we don't have a word.

We haven't talked about how to encode tries in prolog, but we can actually use predicates to encode them. I have a predicate `trie(Pre,C,Post)` which is true if `Pre` is a node in the trie, `C` is a branch from that node, and `Post` is the node by following branch C. For example, we might have `trie(pan,t,pant)`.

We could represent the `p` branch of our trie like this.

```
trie(e0,p,p).
trie(p,a,pa).
trie(pa,n,pan).
trie(pa,s,pas).
trie(pa,t,pat).
trie(pan,t,pant).
trie(pant,s,pants).
trie(pa,s,pas).
trie(pas,t,past).
trie(pa,t,pat).
```

With this in mind, lets take another look at our `make_word` predicate.

```
make_word(Tiles,Part,Word).
```

A more algorithmic way to look at this is that `Tiles` is the list of tiles we have left, and `Part` is our current node in the trie. So our algorithm is

- if we're at a word in the trie, then we're done
  - you can use `word` to see if we've found a word.
- Otherwise remove an arbitrary letter from our list of tiles
- move along the trie with that letter
- make the rest of the word, and put our letter at the front of the word.

For the second part I want to pick a specific tile turn every other letter in the word into a space. I don't have a good reason for doing this yet, but it's going to be important! For example, if I have the word `[p,a,n,t,s]`, and I pick the letter `t`, then I should get back `[' ',' ',' ',t,' ']`. This is the purpose of the `pattern` predicate.

```
?- pattern([p,a,n,t,s],t,P).
P = [' ',' ',' ',t,' '] .
```

If that letter appears multiple times in the word, you may get a few answers back. That's ok, that's actually what we want.

```
?- pattern([b,a,b,o,o,n],b,P)
P = [' ', ' ', b, ' ', ' ', ' '] ;
P = [b, ' ', ' ', ' ', ' ', ' '] ;
false.
```

For this assignment you only need to turn in `ai.pl`

Assignment 11: Throwing tiles on the board

For this final assignment, we're going all the way back to the first assignment. In the first assignment I gave you a word and a position, and I had you place that word on the board.

This time we're doing exactly the same thing. But now we can cheat with prolog! I have the predicate

```
place_word(R,C,Dir,Word,Pattern,Board,NewBoard).
```

- `R` and `C` are numbers between 0 and 14
- `Dir` is either `right` or `down`
- `Word` is a list of letters in a valid word
- `Pattern` is the pattern of the word in `Word`
- `Board` is the current scrabble board represented as a 2D List.
- `NewBoard` is `Board`, but where `Word` have been placed at `R,C,Dir`.

The trick here is that I'm only going to pass in `Word, Pattern`, and `Board`. I don't actually know `R`, `C`, or `Dir`. It turns out that this doesn't really matter. The process is the same.

I'll describe the case for inserting a word where `Dir` happens to be `right`.

- get an arbitrary `Row` from the board.
  - You can pick an arbitrary row by splitting the list into 3 parts. The before part, the single `Row`, and the after part.
- Make sure that `Row` is at position `R` in the list.
- split the `Row` into 3 lists.
- the second list should match the `Pattern`.
  - remember, the `Pattern` is just a list of spaces with a single tile. That means there is enough space on the board for us to put the word, where the letter that isn't in our tiles is in the right place.
- replace the `Pattern` with our `Word` to make a new row.
- put that row back in the board to make our new board.

Once you've completed the `place_word` predicate you can put everything together with the `run` or `run_verbose` predicate. Both of these predicates do the same thing. They generate all of the possible words that can be placed on the board. The `run_verbose` option will print out the board after having placed the word there. In the `scrabble.pl` file, you'll find a predicate `testboard(B)`. This will fill in `B` with our test board. So, to run the program we can use.

```
?- testboard(B), run([p,a,n,t,s], B).
```

.

Finally, to turn in assignment 11, you just need to submit the `ai.pl` file.

Hints

- There are 2 versions of append. We can append 2 lists together, or we can append a list of lists into a single list.
- Placing words going down seems much harder. Can me make this into a problem we've already solved?
- You can constrain the length of a list even if you don't know what's inside the list yet.
- I have given you 2 helpful predicates.
  - `valid_board(B)` is true if all of the words in `B` are valid words.
  - `transpose(B)` transposes the board.
- This part of the assignment took me 13 lines. If you are writing substantially more than that, you may be doing too much work.