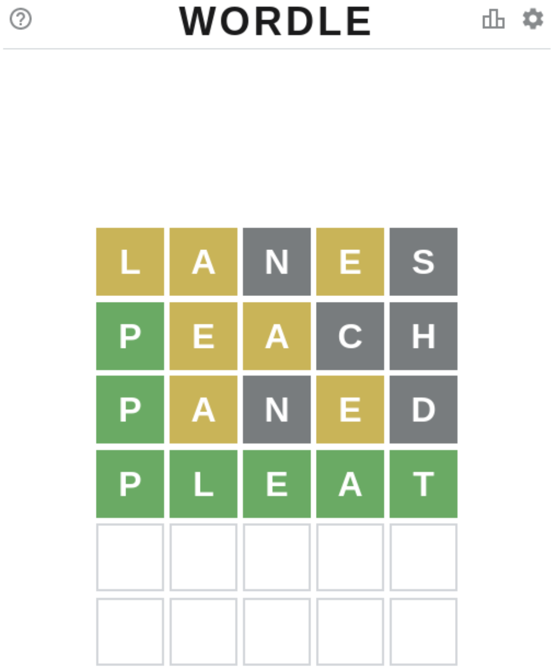
Wordle: functional problem solving



🟨 L	🟨 A	⬛ N	🟨 E	⬛ S
🟩 P	🟨 E	🟨 A	⬛ C	⬛ H
🟩 P	🟨 A	⬛ N	🟨 E	⬛ D
🟩 P	🟩 L	🟩 E	🟩 A	🟩 T

The code

This lecture is based on https:github.com/jimburton/hordle

These slides and the incrementally developed code in them is at https:github.com/jimburton/wordle-lecture

Wordle is (was?) all the rage

Wordle is a game in which you have to guess a secret five-letter word in six tries.

You can only play once a day, and everyone has the same secret word.

It went viral in 2021, partly because there is a neat way of sharing your score without giving away the word.

Scoring

Implementing it

To demonstrate **functional problem solving** in Haskell, we'll do two things:

- implement a version of the game,
- write a solver for it.

Functional problem solving

Haskell offers some distinctive ways of solving problems:

- using **higher-order functions** (e.g. map, `filter`, `foldl`) to create higher-level declarative abstractions (say **what to do** not **how to do it**),
- writing functions using **pattern matching**,
- keeping the majority of the code **pure** (no IO) so that it is easier to test and debug, while having a small **impure** part that does whatever IO is necessary.

Text

We're going to be working with words and lists of words, obviously.

Rather than using `String` (linked lists of `Char` values) we will use `Data.Text`, a much more efficient way to represent text.

`Data.Text` contains lots of functions with names that clash with the `Prelude`, like `map` and `filter`, so we import it qualified except for the constructor:

We enable `OverloadedStrings` so the compiler can treat literals in quotes as `Text` values.

```haskell
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T
import           Data.Text (Text)
```

Getting started

To think about solving our first problem, implementing the interactive game, we could start:

- **top-down** – e.g. the UI, or
- **bottom-up** – small functions that do one thing, like comparing a guess to a secret word.

When *thinking* about what I need to do I might begin top-down, but for *implementation* it's normally bottom-up.

First tasks

So let's start with comparing a guess to a secret word. The first thing we need is a way to represent the score of a letter in the guess:

Each letter has to be marked **green**, **yellow** or **black**.

If a letter is black, it isn't anywhere in the word.

If it is yellow, that *is* context-sensitive, because we are told where it *isn't*.

Similarly with green.

A datatype for scoring

”First” attempt

```
data CharInfo =
    Green [Int]  -- ^ Char is at these indices.
  | Yellow [Int] -- ^ Char is in the target word but not at an
  | Black        -- ^ Char is not in the target word.
  deriving (Show, Eq)
```

(Full disclosure: I began with `Green Int`, then I realised words could have more than one occurence of the same letter.)

A datatype for scoring

But we don't want any duplicates in the lists of positions. So every time we add to the list we would have to check for that. Better to use the right data structure in the first place, which is `Data.Set`.

```
import qualified Data.Set as S
import           Data.Set (Set)

data CharInfo =
    Green (Set Int)  -- ^ Char is at these indices.
  | Yellow (Set Int) -- ^ Char is in the target word but not a
  | Black            -- ^ Char is not in the target word.
  deriving (Show, Eq)
```

Scoring a word

Now we want to take a guess and score it against the secret word. The result will be a list of `(Char, CharInfo)` pairs. We will be dealing with a lot of these lists, so we give them an alias.

As usual, write down the type first.

```haskell
type Guess = [(Char, CharInfo)]

-- | Set the status of each char in a guess.
score :: Text  -- ^ The attempt.
      -> Text  -- ^ The target word.
      -> Guess -- ^ The scored attempt.
score attempt target = undefined
```

Scoring a word

A starting point is to begin by **zipping** the attempt and target up. The zip function in the Prelude works with lists, but there is a version of zip in the text module.

```
ghci> :t T.zip
T.zip :: Text -> Text -> [(Char, Char)]
ghci> T.zip "BEACH" "BEEPS"
[('B','P'),('E','E'),('A','E'),('C','P'),('H','S')]
```

Scoring a word

Now we can map over the zipped up list making comparisons.

For `Green` and `Yellow` we need to know the index of the occurence of this letter, so we zip the list again with the integers from zero, using the ordinary zip from the `Prelude`.

```
ghci> zip it [0..]
[(('B','B'),0),(('E','E'),1),(('A','E'),2),(('C','P'),3),(('H'
```

Scoring a word

The `singleton` function takes a value and produces a `Set` containing just that value.

```
score :: Text  -- ^ The attempt.
      -> Text  -- ^ The target word.
      -> Guess -- ^ The scored attempt.
score attempt target =
  map (\((c,d),i) -> if c==d
                     then (c, Green (S.singleton i))
                     else if T.elem c target
                          then (c, Yellow (S.singleton i))
                          else (c, Black)) $ zip (T.zip attemp
```

Scoring a word

I wrote the funciton on the previous slide but then got a hint from **hlint** reminding me that I could have used `zipWith` (saving me from traversing the lists twice).

```
score :: Text  -- ^ The attempt.
      -> Text  -- ^ The target word.
      -> Guess -- ^ The scored attempt.
score attempt target =
  zipWith (\(c,d) i -> if c==d
                        then (c, Green (S.singleton i))
                        else if T.elem c target
                             then (c, Yellow (S.singleton i))
                             else (c, Black)) (T.zip attempt ta
```

Demo: cabal repl stage1

Scoring a word

A Guess is correct if all the CharInfo values in it are green. We could map isGreen over a Guess then check that all the resulting values are True, but this is what the all function does.

```
isGreen :: CharInfo -> Bool
isGreen (Green _) = True
isGreen _         = False

correct :: Guess -> Bool
correct gs = all (isGreen . snd) gs
```

Scoring a word

And for neatness, let's rewrite `correct` using **pointsfree** style.

```
-- correct gs = all (isGreen . snd) gs is the same as:
correct = all (isGreen . snd)
```

Making more than one guess

Playing a game will mean making guesses until one of them is correct or six attempts have been made.

The neatest way to do this will be by wrapping up what we need to know about a game in a **record**. We will store the previous attempts, as we'll need them later.

```
data Game = Game
  { _word        :: Text       -- ^ The word to guess.
  , _numAttempts :: Int        -- ^ The number of attempts.
  , _attempts    :: [Guess]    -- ^ Previous attempts.
  , _guess       :: Maybe Text -- ^ The latest guess.
  , _done        :: Bool       -- ^ game over flag.
  , _success     :: Bool       -- ^ Game was won.
  } deriving (Show)
```

Records

We can create a game called g by typing `g = Game { _word="BLAH", _numAttempts=0 ... }` and so on, supplying values for all the fields.

If we have a game, g, we can get one of its fields by using the accessor function that is created for the field name:

```
ghci> _word g
"BLAH"
```

Records

We can set one of the fields by giving `key=value` pairs inside braces like this:

```
ghci> g { _word="FOO" , _done=True}
Game { _word="FOO", -- etc
```

Working with games

```
emptyGame :: Game
emptyGame = Game {
    _word = ""
  , _ numAttempts = 0
  , _attempts = []
  , _guess = Nothing
  , _done = False
  , _success = False
  }

gameWithWord :: Text -> Game
gameWithWord secret = emptyGame { _word = secret }
```

Making guesses

```haskell
doGuess :: Game -> Text -> Game
doGuess g attempt = let sc = score attempt (_word g)
                        wn = correct sc
                        dn = wn || (_numAttempts g) == 6 in
  g { _numAttempts = (_numAttempts g)+1
    , _attempts = sc : (_attempts g)
    , _guess = Just attempt
    , _done = dn
    , _success = wn }
```

Note that this is pretty clumsy syntax!

Lenses

This is why we named the record fields with underscores – we want to use the **lenses** library to automatically create convenient getters and setters for `Game`.

I won't go into lenses here, but using them means we can rewrite `doGuess` like this:

```
doGuess :: Game -> Text -> Game
doGuess g attempt = let sc = score attempt (g ^. word)
                        wn = correct sc
                        dn = wn || (g ^. numAttempts) == 6 in
  g & numAttempts %~ +1
    & attempts %~ (sc:)
    & guess ?~ attempt
    & done .~ dn
    & success .~ wn
```

Demo cabal repl stage2

Common lens operators

Operator	Name	Example
(^.)	view	g ^. word: gets word in g.
(.~)	set	g & word .~ ”HELLO”: sets the word of g to ”HELLO”.
(%~)	over	g & word %~ T.toUpper: applies T.toUpper to the word.
(&)	apply	Reverse application, used for supplying the first record to a composed lens, and for chaining operations

Creating the UI

Now that we can take a game, apply a guess to it and check whether the game is over, we need a way to take guesses from the user until that is the case.

We will do this with a simple command-line interface (CLI).

We need an entry point for the game, which will be the usual `main` IO action, and a function that takes guesses until the game is over.

Creating the UI

A top-down outline of what we need, mentioning several functions we haven't written yet:

```
main :: IO ()
main = do
  g <- initGame -- start with a random word
  playGame g

playGame :: Game -> IO ()
playGame = do
  if g ^. done
  then endGame -- tell the user the result
  else do attempt <- getGuess -- get a guess from the user
          let g' = doGuess g attempt
          drawGame g' -- present the result to the user
          playGame () -- take the next move
```

Words and random words

OK, how do we start a game with a random word?

First of all, we need a list of five-letter words to choose from.

Actually, Wordle uses two lists – a short one (∼2300 words), which target words are taken from, and a more complete one (∼12,000 words) that all guesses have to come from.

Having found files containing these lists of words online we can load them in to a list of `Text` values.

Loading dictionaries

This code reads from the file ”etc/long.txt”, splits the resulting string into parts separated by the newline character then makes all words uppercase.

```haskell
-- | A dictionary of five letter words.
dict :: IO [Text]
dict = do txt <- TIO.readFile "etc/long.txt"
          let ls = T.lines txt
          pure (map T.toUpper ls)
```

Loading dictionaries

But if you remember the lecture on the `Functor` typeclass, and recall that `IO` is a `Functor`, we can rewrite this as a oneliner.

```
dict = map T.toUpper . T.lines <$> TIO.readFile "etc/long.txt"
```

Data.Vector

Just as plain old lists weren't the best choice for storing collections of indices, they aren't great for our lists of words.

`Data.Vector` is more like an array than a linked list, so we'll use that.

We make a general action for reading a file into a vector of words to save duplication.

```haskell
import qualified Data.Vector as V

filepathToDict :: FilePath -> IO (Vector Text)
filepathToDict fp = V.map T.toUpper . V.fromList . T.lines <$>

dict :: IO (Vector Text)
dict = filepathToDict "etc/long.txt"

targets :: IO (Vector Text)
targets = filepathToDict "etc/short.txt"
```

Picking a starting word

Now that we have a vector of words we can pick a random value from it.

```haskell
-- | Get a word to be the target for a game.
getTarget :: IO Text
getTarget = do
  flw <- targets
  getStdRandom (randomR (0, length flw)) <&> (V.!) flw

initGame :: IO Game
initGame = getTarget <&> initGameWithWord
```

The UI

With that in place we just need a CLI that loops asking the user to enter a word.

To make entering text in a terminal more convenient, we use the ‘haskeline‘ library. It enables the backspace and arrow keys, among other things.

It runs in its own monad so to carry out IO actions inside it we use `liftIO`.

The UI

Demo cabal run stage 3

Making an automated solver

In the game the player, if they are any good, chooses the next guess based on the feedback from previous ones.

We need a way of organising the feedback from the game and using it to filter the dictionary.

This can be used to provide hints to a human player but also to solve entire games.

The info map

We will have some information on every letter that has appeared in a guess so far, and that information will be encoded as a `CharInfo`.

It stands to reason that no letter can have more than one colour, so what we want is a **map** from `Char` to `CharInfo`.

We will need to add to this map after each go, and possibly change the status of letters.

For instance it could be that all we knew about a letter was that it was in the word somewhere (yellow) but find its position on the next guess, so its status goes from yellow to green. Status will never change from green or black though.

The info map

We add the map to the game record.

```haskell
import           Data.Map (Map)
import qualified Data.Map as Map

data Game = Game
  { -- ...
  , _info     :: Map Char CharInfo -- ^ Info on previous guess
    -- ...
  } deriving (Show)
```

We make a new map with `Map.empty` and add it to the initial game.

Updating the map

Now we can go back to doGuess and refactor it to update the map too.

```
doGuess :: Game -> Text -> Game
doGuess g attempt = let sc = score attempt (g ^. word)
                        wn = correct sc
                        dn = wn || (g ^. numAttempts) == 5 in
  g & numAttempts %~ +1
    -- ...
    & info %~ updateMapWithAttempt sc
```

Updating the map

The type of the function that updates the map with a new guess:

```
updateMapWithAttempt :: Guess -> Map Char CharInfo -> Map Char
```

It is simple enough but too long for a slide. There are three cases for every `(Char, CharInfo)` pair. `(c, ci)`, in the guess:

- `ci` is green or yellow: if `c` is already in the map, update its value with the index `i`, otherwise add it,
- `ci` is black: add `c` to the map, not worrying about whether we overwrite anything.

Using the map

Each piece of information, `(c, ci)`, in the map represents a constraint on the next best guess. We can find candidate words by filtering the dictionary down to a new list, `d'`, so that

1. If `ci` is of the form `Green s`, `d'` contains only words that have c at all of the indices in the set `s`.
2. If `ci` is of the form `Yellow s`, `d'` contains only words that contain c at least once but not at any of the indices in `s`.
3. If `ci` is `Black` `d'` contains only words that do not contain c.

See `findWords`

Hints

As the game progresses the list of constraints grows and the list of candidates shrinks rapidly.

We add a ":HINT" command to the CLI. It selects a single word from the list of candidates.

It uses the functions `hints`, which selects all candidate words for the constraints, and `hint`, which just picks the first one in the list.

Demo cabal run stage4

The solver

Finally, we want to use `hint` to play whole games from scratch.

For the first word, there is no information available. Hints would include the entire dictionary.

We pick one of the words linguists have identified as a good choice for a starting guess: **SOARE** (which means a young eagle).

Can you imagine why they suggest it's a good choice?

The solver

After submitting our first guess we start to get feedback from the game, which we can use to get a much smaller set of hints.

Each of these candidate words, when applied as the next guess, will in turn lead to a narrowed down list of possible candidates.

We could try applying eah candidate word, then for each of the subsequent candidates, try applying them, and so on, until we get the answer.

This *brute force* seach would take way too long to run.

The solver

Our strategy is to *choose a word, w, that leads to the fewest possible subsequent candidates*.

So we get the list of possible candidates for the given constraints, apply each of them as a guess and look at how many candidates there are for the guess after that.

```
hint g = do
  hs <- hints g -- gets all candidate words for the current co
  let possibleGames = V.map (\t -> (t, doGuess g t)) hs -- mak
  reds' <- mapM (\(t,g') -> hints g' <&> (t,) . length) possib
  let res = sortBy (\(_,l1) (_,l2) -> l1 `compare` l2) $ V.toL
  pure $ fst <$> listToMaybe res
```

The solver

This is a **greedy** algorithm, in that it takes what looks like the best choice based just on the local information.

But it may not actually be the best choice – it could lead to a dead end.

So we have to build in some **backtracking**. If w turns out to be a dead end we will retrace our steps.

Automating the solver

We finish by macking a number of functions that automate the playing of a game.

The functions are `solve`, `solveTurn` and `backtrack`. The algorithm:

- Start a game with the fixed word.
- LOOP (until the game is over):
    - IF there is a next best hint, play it and continue
    - ELSE undo the last guess, blacklist it and try another word

It is a pretty simple approach, and could be made ot run a lot faster, but it can solve any word in the Wordle list in an **average of 2.7 guesses** :-)

Demo cabal repl stage5