2023 Haskell January Test
Cryptic Clue Solving

This test comprises four parts (Part IV is unassessed) and the maximum mark is 30.

There will be a special bonus prize for the first student to complete all four Parts and whose code passes all the tests in the given test suite.

The **2023 Haskell Programming Prize** will be awarded for the best overall solution, or solutions.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.

1 Introduction

A traditional crossword clue typically comprises the *definition* of some word or phrase which has to be guessed by the solver. The number of letters in the solution, also called the *enumeration*, is also part of the clue, so the length of any guessed answer must match the enumeration. For example, given the clue

Pardon (7)

the solver might guess solutions such as "forgive", "condone", "absolve" etc., all of which mean "pardon" and have the required seven letters.

In contrast, a *cryptic* crossword clue typically has two components: a traditional crossword-like definition and a separate *wordplay* which is an independent set of instructions, encoded in natural language, for how to assemble the solution from fragments of text contained within, or derived from, the clue. For example, in the correct reading of the cryptic clue:

Pardon girl ringing home (7)

the definition is "pardon" (as above) and the wordplay is "girl ringing home". The trick is to ignore the surface reading of the wordplay and think of it instead as a code – in this case it's telling us that the solution comprises the letters of a synonym for "girl" surrounding (i.e. "ringing") the letters of a synonym for "home". In this case the "girl" refers to "amy" (a girl's name) and "home" to "nest", hence the solution "am nest y" – a word meaning "pardon". The word "ringing" in the clue is called an *indicator*, which, in this case, is the encoded instruction to put one fragment of text inside another. We will refer to this as an *insertion*. In this exercise you can think of insertions as infix operations where the two arguments are sub-clues that specify the text fragments upon which the insertion operates.

In addition to insertions there are around 20 other operations commonly used by setters, but in this exercise we'll focus on the following:

Synonyms

A synonym of a word or phrase is a (single) word which has the same meaning, e.g. "massive" → "huge", "maker of bread" → "baker" (or possibly "earner", if we take "bread" to mean "money"!). Synonyms are unindicated, so the word "sort" for example, could refer to the verbatim text "sort" or to another word with the same meaning, e.g. "order". The simplest type of cryptic clue type is a so-called *double definition* involving just the replacement of one word by a synonym, as in:

Business worry (7)

which has at least one solution: "concern" – a word that means both "business" and "worry". Note that it's not clear which of these two words is the definition and which is the wordplay: both interpretations resolve to the same answer, so there are arguably two solutions, as there are two (different) ways to construct the same answer. In fact, the word "problem" can also be seen as a synonym of both "business" and "worry", in which case there are arguably four solutions; there may be many more, depending on whether other similar synonyms can be found.

Anagrams

Anagrams involve rearranging the letters of one or more words. Anagram indicators are words or short phrases that suggest the notion of rearrangement, scrambling, mixing up etc. For example, in clue

New toaster revolves (7)

the word "new" suggests making a new version of the letters of "toaster", ending up with a word than means the same as "revolves". The answer is "rotates". Notice that in this case

the definition ("revolves") appears at the end of the clue, c.f. "pardon" above which is at the beginning; definitions can appear at either end.

The argument text for an anagram will always be literal text appearing verbatim in the clue here, as in the text "toaster" above[1].

Reversals

Reversals involve reversing the letters of a fragment of text and are indicated by words/phrases such as "about", "backwards", "in reverse" etc. For example, in the clue:

Cheese made from east to west (4)

the phrase "from east to west" (implying writing the letters out from right to left) is the reversal indicator and the letters to reverse are those of "made". The answer is "edam" – a type of cheese. In this case the text to reverse appears verbatim in the clue. However, unlike anagrams, the argument can be any nested construction (presumably solved by recursion), e.g. from the evaluation of a synonym of another word or phrase, as in:

Plans to return junk mail (4)

Here, "to return" is the reversal indicator and the target text (the argument) is "junk mail" which must first be replaced by "spam" (a synonym of "junk mail"). The answer is "maps" (the letters of "spam" written in reverse), which is an alternative word for "plans".

Insertions

An example of an insertion is given above. As with reversals, there are no constraints on the two arguments, i.e. either could involve a synonym, an anagram, a reversal, a charade (next rule), or even another insertion.

Note that the insertion in the above clue is sometimes referred to as an *envelope* in the sense that the second argument ("home" → "nest") defines the text to be inserted and the first argument ("girl" → "amy") defines the target of the insertion, which ends up *enveloping* the insertion text. A "standard" insertion has the arguments the other way round, as in:

River, one featured in superstitions (5)

where "featured in" is an insertion indicator whose *first* argument ("one" → "i") defines the text to be inserted and whose second ("superstitions" → "lore") defines the target text; the answer is thus "lo i re" – the name of a river.

Charades

A charade is simply a concatenation of two text fragments, each derived (recursively) from some other construction, including a nested charade. For example, the clue

Taxi home from hut (5)

has the solution to "cabin" (a synonym of "hut"), with the wordplay being a charade of "taxi" → "cab" and "home" → "in" (in the sense of "being home"). The word "from" here is a so-called *link* that separates the definition and wordplay. Link words/phrases are optional.

In the above clue the charade is unindicated. A variation allows infix indicators such as "and", "next to", "with" etc., as in:

One with hand that's perfect (5)

which has the solution "ideal" – a charade of "one" → "i" and "hand" → "deal". The word "with" is the charade indicator and "that's" is a link word.

[1]This convention is followed religiously by most setters with so-called "indirect anagrams", where a word or phrase is replaced by a synonym before being scrambled, being deemed unfair.

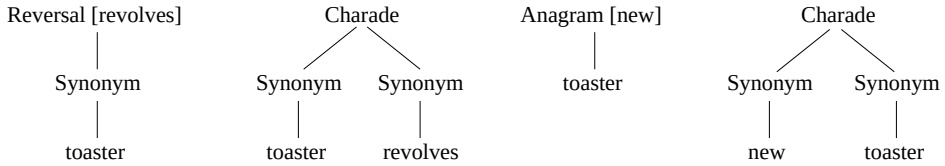

Figure 1: Parse trees for "New toaster revolves (7)"

2 Clue solving in Haskell

Automating the task of clue solving involves:

- Parsing a clue in all possible ways, with each possible interpretation being parsed as a *definition*, an optional *link* word/phrase and a *parse tree* which represents the various operations encoded in the wordplay.
- Evaluating the parses in order, each generating zero or more result strings. Every result string produced is deemed to be a valid solution – there may be more than one and in some cases the same string may be produced multiple times.

2.1 Clue parsing

Because clues are written using natural language there are typically many parses of a clue. For example, the above clue:

New toaster revolves (7)

might have the following four interpretations:

1. A reversal ("revolves") of "toaster", with definition "new".
2. A charade of "toaster" and "revolves", with definition "new".
3. An anagram ("new") of "toaster", with definition "revolves".
4. A charade of "new" and "toaster", with definition "revolves"

The number of readings of the clue depends on whether the component words/phrases can be interpreted as indicators. For example, if we decide that "revolves" should *not* be interpreted as a reversal indicator then there might be just three parses instead of four and if "toaster" is treated as an anagram indicator then there may be as many as six.

The parse trees for these four interpretations are shown in Figure 1. The indicator for each operation (if there is one) is placed in [square brackets] next to the operation at each node. Notice that synonyms always appear at the leaves of the tree and that the argument of an anagram is always verbatim text (it is not parsed as a synonym).

Note that having parsed a clue we have no idea in advance which, if any, of the parses will lead to a solution – that depends on how the various words and phrases get mapped to synonyms and how their component letters are manipulated by the various operations – this is the responsibility of the evaluator (see below). It is therefore necessary to evaluate *all* valid parses in an effort to find a solution.

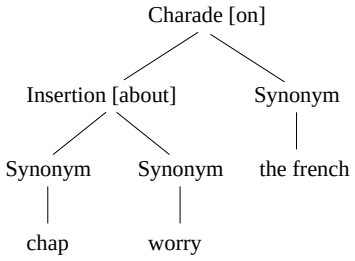

Figure 2: Correct parse tree for "Worry about chap on the French port (9)"

2.2 Recursive constructions

The argument(s) of the insertion, reversal and charade operations can be arbitrary "nested" sub-clues, which means that clue parsing and clue evaluation are inherently recursive in nature. In principle, the depth of nesting is unbounded, although clues tend to be short, which limits the amount of nesting seen in practice. As an example,

Worry about chap on the French port (9)

has the solution "fremantle" (an Australian port) and the correct reading of the wordplay is a charade of: a. An insertion of "chap" → "man" into "worry" → "fret" and b. "the" → "le" (the French word for "the"); the two indicators are "about" (envelope form of insertion) and "on" (charade). There are thus two levels of operation nesting, as show in the "correct" parse tree, i.e. the one that resolves to the answer, in Figure 2.

Note that a charade of three or more text fragments can be constructed by nesting two-way charades accordingly, in the same way that three numbers, x, y and z, can be added together using binary addition, viz. $x + (y + z)$ or $(x + y) + z$. The fact that concatenation is associative means that there may be two ways to construct a three-way charade when one will do, but we will ignore that for the purposes of the exercise; the effect will be that two or more parse trees might evaluate to give the same result string.

2.3 Representation

The following data types for representing clues and their parses is given in the `Types` module (note that definitions, indicators and links may comprise multiple words, hence their representations as *lists* of strings):

```
type Clue = (String, Int)

type Definition = [String]

type Indicator = [String]

type Link = [String]

data ParseTree = Synonym String |
                 Anagram Indicator String |
                 Reversal Indicator ParseTree |
                 Insertion Indicator ParseTree ParseTree |
                 Charade Indicator ParseTree ParseTree
```

```
                deriving (Show)

type Parse = (Definition, Link, ParseTree)
```

There is an additional `ParseTree` constructor in the `Types` module, but this is only relevant for Part IV, which is unassessed.

For example, the "fremantle" clue above will be represented by `("Worry about chap on the French port",9)` and the `Parse` that leads to the correct solution is (formatted to aid readability):

```
(["port"],[],Charade ["on"] (Insertion ["about"] (Synonym "chap")
                                                 (Synonym "worry"))
                            (Synonym "the french"))
```

Notice that there is no link separating the wordplay and definition in this clue, hence the `[]`.

We will assume throughout that the first argument of the `Insertion` constructor defines the text to be inserted, here "chap" → "man", and the second the target of the insertion, here "worry" → "fret".

2.4 Evaluation

There are several ways one might go about evaluating a parsed clue. In this exercise you'll build a *backwards* evaluator which starts by using the `Definition` to guess a solution and then determining whether the guess *matches* the corresponding `ParseTree`. Each definition may have many synonyms (guesses), and any number of them might match the parse tree, so you should try to match them all, in order.

Synonym mapping and indicator recognition are clearly crucial to any solver. To simplify the task you are provided with a collection of *static* data sets from which to build a solver: a *thesaurus* which contains a list of synonyms of around 2000 common words and phrases, and a list of indicator words and phrases for each of the indicated operations (anagram, reversal, insertion and charade). Access to the thesaurus is provided via a function `synonyms :: String -> [String]` in the module `WordData.hs`. For example:

```
Solver> synonyms "ornate"
["detailed","fine","grand","ornate"]
```

Note that every word has itself as one of its synonyms.

The indicators for each operation are provided in a suitably named constant in the same module, e.g.

```
*Solver> reversalIndicators
["about","after recovery","back","backed","backward",...]
```

You'll need the indicator lists in Part III, but not Parts I and II.

The rules for matching a string, `s`, against a parse tree are as follows:

- If the tree is a `Synonym` of some string `s'` then the match succeeds if the list of `synonyms` of `s'` contains `s`.
- If the tree is an `Anagram` of the verbatim text `s'` then the match succeeds if `s` and `s'` contain the same letters. The order is unimportant, so you can solve the problem by asking whether the *sorted* versions of `s` and `s'` are the same.
- If the tree is a `Reversal` of some subtree `t` then the match succeeds if the reversal of `s` matches `t`.

- If the tree is an `Insertion` with argument subtrees `t1` and `t2` then you need to decompose `s` into all pairs of non-empty strings, `s1` and `s2`, say, such that the insertion of `s1` into `s2` yields `s`. The match then succeeds provided `s1` matches `t1` and `s2` matches `t2`. For example, if `s = "ache"` then the valid decompositions are `("c","ahe")`, `("ch","ae")` and `("h","ace")`; you can think of this decomposition being a process of "uninsertion".
- If the tree is a `Charade` with argument trees `t1` and `t2` then you need to split `s` into all pairs of strings, `s1` and `s2`, say, such that `s1 ++ s2 == s`. Similar to insertions, the match succeeds provided `s1` matches `t1` and `s2` matches `t2`.

Given a parse `p = (def, link, t)` of some clue `c`, a valid solution is a synonym, `s` say, of the definition, `def`, that matches the parse tree `t`; this is returned as a component of a triple of `type Solution = (Clue, Parse, String)`, viz. `(c, p, s)`. For example, for the "correct" parse of the "fremantle" clue above `def = "port"`, `s = "fremantle"` and `t` is the representation of the parse tree in Figure 2, as shown in Section 2.3. The `Solution` type synonym above is included in the `Types` module.

3 What to do

3.1 Part I: Preliminaries

1. When solving a cryptic clue all punctuation can be removed before starting, as it's invariably "bluff". Hence, define a function `cleanUp :: String -> String` that will remove all punctuation from a string and convert all upper-case characters to lower case (use `toLower` from `Data.Char`). The punctuation characters are defined by the constant `punctuation = "';.,-!?"` in the template. For example:

```
*Solver> cleanUp "half-baked"
"halfbaked"
*Solver> cleanUp "Haskell - everyone's favourite language!"
"haskell  everyones favourite language"
```

[1 Mark]

2. Define a function `split2 :: [a] -> [([a], [a])]` that will split a list into all pairs of non-empty sublists which, when concatenated (`++`) reconstruct the original list. For example,

```
*Solver> split2 ""
[]
*Solver> split2 [1,2,3,4]
[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]
```

Hint: You might like to use functions like `length`, `splitAt` etc. (it's an $O(n^2)$ problem). The order of the elements in the resulting list is unimportant.

[2 Marks]

3. Define a function `split3 :: [a] -> [([a], [a], [a])]` that will split a given list into all triples of sublists which, when concatenated (`++`) reconstruct the original list. The first and third elements must be non-empty, but the middle element can be null (`[]`). For example,

```
*Solver> words "a sample word play"
["a","sample","word","play"]
*Solver> split3 (words "a sample word play")
[(["a"],["sample"],["word","play"]),
 (["a"],["sample","word"],["play"]),
 (["a","sample"],["word"],["play"]),
 (["a"],[],["sample","word","play"]),
 (["a","sample"],[],["word","play"]),
 (["a","sample","word"],[],["play"])
]
```

The order of the elements in the resulting list is unimportant.

Hint: Use `split2`: once to split the original list into a list of pairs. For each pair, `(xs1, xs2)`, say, split `xs1` (or, equivalently `xs2`) similarly and use the results to form the required triples of non-empty lists. You can re-use `xs1` and `xs2` to build those triples with null middle elements.

[2 Marks]

4. Using `split3`, or otherwise, define a function `uninsert :: [a] -> [([a], [a])]` that will decompose a list into all pairs of non-empty sublists, `xs1` and `xs2`, say, such that the original list can be reconstructed by inserting `xs1` into `xs2` at some point. For example:

```
*Solver> uninsert "abcde"
[("b","acde"),("bc","ade"),("bcd","ae"),("c","abde"),
 ("cd","abe"),("d","abce")]
*Solver> uninsert [1,2]
[]
```

The order of the elements in the resulting list is unimportant.

[2 Marks]

Mirrored splits

Later on you'll see that it's useful to have versions of `split2` and `split3` that include "mirror images" of the results of the original functions, suitable named by appending "`M`" to their names, e.g.

```
*Solver> split2M [1,2,3]
[([1],[2,3]),([1,2],[3]),([2,3],[1]),([3],[1,2])]
*Solver> split3M "abcd"
[("a","b","cd"),("a","bc","d"),("ab","c","d"),("a","","bcd"),
 ("ab","","cd"),("abc","","d"),("cd","b","a"),("d","bc","a"),
 ("d","c","ab"),("bcd","","a"),("cd","","ab"),("d","","abc")]
```

The code for these functions is essentially "boilerplate", so they have been defined for you, but commented out, in the template. Once you've defined `split2` and `split3` you should uncomment them and make sure that they work as expected.

3.2 Part II: Evaluation

To help you to test these functions, a number of parse trees are provided in the module `Examples` in the form of a list `trees :: [ParseTree]`. A function `showT :: ParseTree -> IO ()` is also provided in the `Types` module that will pretty-print a parse tree, with the five constructions being respectively displayed by the abbreviations `SYN`, `ANAG`, `REV`, `INS`, and `+` (infix) for charades. Where relevant the indicators are shown in [square brackets]. For example,

```
*Solver> trees !! 2
Anagram ["unusual"] "hensdark"
*Solver> showT (trees !! 2)
ANAG[unusual] hensdark

*Solver> trees !! 12
Charade ["on"] (Insertion ["about"] (Synonym "chap") (Synonym "worry"))
               (Synonym "the french")
*Solver> showT (trees !! 12)
INS[about] (SYN chap) (SYN worry) +[on] SYN the french
```

1. Define a function `matches :: String -> ParseTree -> Bool` that implements the matching scheme described in Section 2.4 above. The list `strings :: [(String, String)]` in the `Examples` module contains pairs of strings that respectively do and do not match the corresponding elements of `trees`. For example,

   ```
   *Solver> matches "road" (Synonym "street")
   True
   *Solver> matches "road" (Synonym "apricot")
   False
   *Solver> zipWith matches (map fst strings) trees
   [True,True,True,True,True,True,True,True,True,True,True,True,
    True,True,True,True]
   *Solver> zipWith matches (map snd strings) trees
   [False,False,False,False,False,False,False,False,False,False,
    False,False,False,False,False,False]
   ```

   [7 Marks]

2. Using `matches` define a function `evaluate :: Parse -> Int -> [String]` that, given a clue parse comprising a definition, a link word/phrase (unused here) and a parse tree, will return all synonyms of the definition that match the tree. Note that the definition is represented as a list of words, so you should apply `unwords` before generating the list of synonyms. The integer argument is the enumeration, so, as an optimisation, you can filter out synonyms that do not have the required length.

   For example (`figParses` contains the representations of the four parses whose parse tree feature in Figure 1):

   ```
   *Solver> evaluate (figParses !! 0) 7
   []
   *Solver> evaluate (figParses !! 2) 7
   ["rotates"]
   *Solver> map (flip evaluate 7) figParses
   [[],[],["rotates"],[]]
   ```

Note that only the anagram (parse index 2) resolves to the answer.

The parses for the first 19 clues in the `Clues.hs` module that resolve to give the correct answer are given in `parses :: [Parse]` in the `Examples` module. The corresponding enumerations are given in `enumerations :: [Int]`, so you can further test your `evaluate` function thus:

```
*Solver> zipWith evaluate parses enumerations
[["concern","problem"],["rotates"],["redshank"],["master"],["edam"],
 ["repaid"],["amnesty"],["remainder"],["sustain"],["loire"],["cabin"],
 ["snappy"],["fremantle"],["nasty"],["rotate"],["inapt"],
 ["kensington"],["defiant"],["speed"]]
```

Note that the thesaurus has two synonyms for `business` that are also synonyms of `worry` ("concern" and "problem"), so both are treated as valid solutions (the order is unimportant).

[3 Marks]

4 Part III: Clue parsing

1. Define a parsing rule for each of the five constructions (synonym, anagram, reversal, insertion, charade). In each case the argument is a fragment of clue text (a list of words) and the result is a list of parse trees.

   Optional idea: You can implement the parsing rules in any way you wish, but you may observe from reading below that there is quite a lot of similarity among the one-argument (anagrams and reversals) and two-argument constructions (insertions and charades), so you might find it useful to abstract the corresponding parsing rules as higher-order functions. The rule for synonyms will be a special case, but if you get the higher-order functions right you should be able to express each of the remaining rules as one- or two-liners.

   The template contains a type signature for each parsing rule and a definition for a function `parseWordplay` that combines them all together:

   ```
   parseWordplay :: [String] -> [ParseTree]
   parseWordplay ws
     = concat [parseSynonym ws,
               parseAnagram ws,
               parseReversal ws,
               parseInsertion ws,
               parseCharade ws
              ]
   ```

   For example, in the above clue "New toaster revolves (7)" the wordplay text, `ws`, will initially be either be the list of words `["new","toaster"]` or `["toaster","revolves"]`, with the remaining word in each case being the definition, which will be handled separately (see below). The template has been set up with appropriately defined defaults for each parsing function so that you can define and test each in turn before moving onto the next. *The order of the elements in the lists generated by each parsing rule is unimportant.* The wordplay parsing rules are as follows.

Synonyms

Any fragment of text can be parsed as a synonym, provided there is a synonym of the `words` of that text in the thesaurus (use the `synonyms` function to check this), e.g. `"new"` has a synonym, so the text fragment `["new"]` can be parsed to give a singleton parse tree `[Synonym "new"]`. However, there is no synonym of `"new toaster"` so `["new","toaster"]` will return `[]` in this case. Note that the rule returns either one tree or none at all. For example,

```
*Solver> parseSynonym (words "great")
[Synonym "great"]
*Solver> parseSynonym (words "at home")
[Synonym "at home"]
*Solver> parseSynonym (words "triangular mollusc")
[]
```

Anagrams

Anagrams are explicitly indicated and the indicator can appear either to the left or to the right of the argument text, e.g. `["mixed","bag"]` and `["bag","mixed"]` will both be parsed as an anagram of the literal text `"bag"`, because the string `"mixed"` is an element of `anagramIndicators` – see the `WordData` module. To parse an anagram, first split the argument text into two fragments in all possible ways using `split2M` (e.g. `["mixed","bag"]` will be split as `[(["mixed"],["bag"]), (["bag"],["mixed"])]`). Now inspect each pair: if the words of the first fragment form an anagram indicator, `ind` say, then build the singleton parse tree `[Anagram ind arg]` where `arg` is the concatenation (no spaces) of the words of the second fragment; otherwise return `[]`. Note that using `split2M` rather than `split2` means that the left- and right-hand cases can be handled by a single rule, rather than two. For example,

```
*Solver> parseAnagram (words "mixed bag")
[Anagram ["mixed"] "bag"]
*Solver> parseAnagram (words "bag mixed")
[Anagram ["mixed"] "bag"]
*Solver> parseAnagram (words "changed the car")
[Anagram ["changed"] "thecar"]
*Solver> parseAnagram (words "fruit cake")
[]
```

As with synonyms, note that the resulting list is either `[]` or a singleton.

Hint: use `unwords` to convert a candidate indicator into a string, e.g. `unwords ["mixed", "up"]` → `"mixed up"`.

Reversals

The parsing rule for reversals is similar to that of anagrams except that the second fragment of each split pair must be recursively parsed using `parseWordplay` to yield a list of parse trees; each such tree, `t` say, is then used to build a parse tree of the form `Reversal ind t` (`ind` is the indicator), e.g.:

```
*Solver> parseReversal (words "go backwards")
[Reversal ["backwards"] (Synonym "go")]
*Solver> parseReversal (words "backwards go")
[Reversal ["backwards"] (Synonym "go")]
*Solver> parseReversal (words "mixed bag")
[]
```

Note that, in general, any number of trees may be produced; you can test this later (see below).

Insertions

Insertions are assumed to be infix operations, so the input text must first be split into three fragments in all possible ways using `split3`. A split of the form `(arg, ws, arg')` is parsed as an insertion if the words of `ws` form an insertion indicator, `ind`, say. The two arguments `arg` and `arg'` must then be recursively parsed using `parseWordplay` and all pairwise combinations of the elements of the resulting lists used to form the arguments to the `Insertion` in the resulting parse trees.

Importantly, there are two "flavours" of insertion: envelopes, of the form 'X around Y', and standard insertions, of the form 'X within Y' and there are separate indicator lists for each: `envelopeIndicators` and `insertionIndicators` respectively. The parsing rule therefore requires two sub-rules differing in two respects: 1. The list of indicator words/phrases used to check the parse and 2. the order of the two arguments of each `Insertion` if/when the check succeeds. The reason for using `split3` as suggested, rather than `split3M`, for example, is because the argument order is significant. Note also that a split of the form `(arg, [], arg')` will yield an empty list of parses, as required, because the empty string (`unwords []`) is not an envelope/insertion indicator.

```
*Solver> parseInsertion (words "back in business")
[Insertion ["in"] (Synonym "back") (Synonym "business")]
*Solver> parseInsertion (words "work around town")
[Insertion ["around"] (Synonym "town") (Synonym "work")]
*Solver>  parseInsertion (words "back pain")
[]
```

Note the order of the arguments to `Insertion` in the second case (an *envelope*).

Charades

The parsing rule for charades is almost identical to that of insertions, the main difference being the indicator lists used. As with insertions there are two "flavours" of charade: a left-to-right version, which may be indicated or not, i.e. of the form 'X Y' (unindicated concatenation), or 'X before Y', and a right-to-left version of the form 'X after Y'. The corresponding indicator lists are `beforeIndicators` and `afterIndicators`. Note that `""` is an element of `beforeIndicators`, so if there is no indicator (`[]`) then the corresponding three-way split will be correctly parsed as an unindicated charade.

```
*Solver> parseCharade (words "stop go")
[Charade [] (Synonym "stop") (Synonym "go")]
*Solver> parseCharade (words "stop and go")
[Charade ["and"] (Synonym "stop") (Synonym "go"),
 Charade [] (Synonym "stop") (Charade [] (Synonym "and") (Synonym "go")),
 Charade [] (Charade [] (Synonym "stop") (Synonym "and")) (Synonym "go")]
*Solver> parseCharade (words "go after stop")
[Charade [] (Synonym "go") (Charade [] (Synonym "after") (Synonym "stop")),
 Charade [] (Charade [] (Synonym "go") (Synonym "after")) (Synonym "stop"),
 Charade ["after"] (Synonym "stop") (Synonym "go")]
```

Again, note the order of the arguments to `Charade` in the final case. Note also the redundancy referred to earlier when parsing charades as described, as the second and third parses of `"stop and go"` above will will both evaluate to the same result(s). Fixing this is messy, so we will not worry about such redundancy in the exercise.

Now is a good time to re-test your parsing rules for reversals and insertions, e.g.:

```
*Solver> parseReversal (words "back at home")
[Reversal ["back"] (Synonym "at home"),
 Reversal ["back"] (Charade [] (Synonym "at") (Synonym "home"))]
*Solver> parseInsertion (words "was back in time")
[Insertion ["in"] (Reversal ["back"] (Synonym "was")) (Synonym "time"),
 Insertion ["in"] (Charade [] (Synonym "was") (Synonym "back")) (Synonym "time")]
```

[9 Marks]

2. Define a function `parseClueText :: [String] -> [Parse]` that combines the functions `split3M` and `parseWordplay` to parse the text of a given clue (a list of its component words) in all possible ways. This involves

   (a) Splitting the text into a candidate definition, optional link, and wordplay

   (b) Checking that the link word/phrase is recognised as one of the given `linkWords`

   (c) Checking that there is at least one synoynm of the definition (otherwise no solution can ever be found)

   (d) Parsing the wordplay

   in all possible ways. Recall that there may be one or more link words between the definition (`D`) and wordplay (`WP`), but these will assume to "commute", so that a particular link word/phrase can have the definition to the left and the wordplay to the right, or the other way round. For example, `D that is WP` and `WP that is D` are both taken to be valid, because the string `"that is"` is an element of `linkWords`. Hence, you should use the "mirror" version `split3M` for building three-way splits in this case. Importantly, `""` is an element of `linkWords` so, given a split of the form `(D, [], WP)`, for example, `unwords []`, i.e. `""`, will be recognised as a valid link.

   A function `parseClue :: Clue -> [Parse]` is defined in the template that invokes `parseClueText` on the words a of a cleaned-up version of the clue text. You will find that there can be several hundred parses for some clues, so a quick way to check that you've not missed any is to count them. Hence, for example:

```
parseClue clue@(s, n)
  = parseClueText (words (cleanUp s))
```

For example,

```
*Solver> parseClue (clues !! 0)
[(["business"],[],Synonym "worry"),(["worry"],[],Synonym "business")]
*Solver> parseClue (clues !! 1)
[(["new"],[],Reversal ["revolves"] (Synonym "toaster")),
 (["new"],[],Charade [] (Synonym "toaster") (Synonym "revolves")),
 (["revolves"],[],Anagram ["new"] "toaster"),
 (["revolves"],[],Charade [] (Synonym "new") (Synonym "toaster"))]
*Solver> length (parseClue (clues !! 16))
72
```

[3 Marks]

3. Finally, define a function `solve :: Clue -> [Solution]` that will solve a given clue, returning a list of solutions, as there may be more than one. A valid solution comprises the clue, the `Parse` that led to the solution and a valid answer (a `String`); there will thus be one solution for each string generated by evaluate. If there is no solution then the result should be `[]`. The function `showSolutions` in the `Types` module will pretty-print a solution, making the output easier to read, e.g.

```
*Solver> solve (clues !! 1)
[(("New toaster revolves",7),
 (["revolves"],[],Anagram ["new"] "toaster"),
 "rotates")]
*Solver> showSolutions (solve (clues !! 16))

CLUE: Carol, in county, working for palace (10)
Solution: kensington
DEF: palace
LINK: for
TREE: INS[in] (SYN carol) (SYN county + SYN working)

CLUE: Carol, in county, working for palace (10)
Solution: kensington
DEF: palace
LINK: for
TREE: INS[in] (SYN carol) (SYN county) + SYN working
----
```

Note that clue 16 has two structurally distinct parses that resolve to the same (correct) solution, "kensington".

If you've got this far then you should be able to solve the first 19 clues, which you can test via:

```
*Solver> solveAll 19
[["concern","problem"],["rotates"],["redshank"],["master"],["edam"],
```

```
["repaid"],["amnesty"],["remainder"],["sustain"],["loire"],["cabin"],
["snappy"],["fremantle"],["nasty"],["rotate"],["inapt"],
["kensington"],["defiant"],["speed"]]
```

[1 Mark]

5 Part IV

There are no marks for this, so you should only attempt this if you have completed the above.

A "hidden word" is a word that is contained verbatim with one or more other words, e.g. `gillingham` contains the word `ling` and `molecular genetics` contains the word `large`, which spans both words. The rule is that the hidden word must be strictly within the target text (it must not include the first and last letters of the target text) and it must span all the words in the target text, e.g. `ham` is not a hidden word in either `gillingham` or `gillingham town`.

Add an evaluation rule and parsing rule for hidden words to your solver. A constructor `HiddenWord` is already included in the `ParseTree` data type, with a corresponding rule in the `show` functions. The hidden word indicators are provided in `hiddenWordIndicators`. For example:

```
*Solver> clues !! 19
("Fish from Gillingham",4)
*Solver> showSolutions (solve (clues !! 19))
CLUE: Fish from Gillingham (4)
Solution: ling
DEF: fish
TREE: HW[from] gillingham
----

*Solver> clues !! 20
("Boy hiding in fitted wardrobe",6)
*Solver> showSolutions (solve (clues !! 20))
CLUE: Boy hiding in fitted wardrobe (6)
Solution: edward
DEF: boy
TREE: HW[hiding in] fitted wardrobe
----
```

There is a special bonus prize for the first student who can solve all 24 clues in the `Clues` module (`solveAll 24`).

Good luck!