diseases

7.3.3 HE SCORED HARLEM WINDS (6) (= "MAHLER") [Guardian]

Not only is knowledge of composer Gustav Mahler required, but also a cryptic understanding that 'HE SCORED' can refer to a member of the set of male composers. Note that this is structurally different from the examples above: while (1) was a more oblique version of a synonym (litter **is** paper on the street), and (2) is membership of the set of of playwrights, we must now consider the set of people who fit the description "he scored", which may include composers, sportsmen, and maybe even engravers.

7.3.4 WHERE AND HOW A SUPERHERO MIGHT LABEL HIS FAUCET (4) (= "BATH") [Guardian]

This clue requires not just specialist knowledge, but also natural language parsing of the sentence of a whole. The answer can be derived from the concept that the superhero Batman would append bat- onto the names of objects (batmobile, etc.), and that a hot tap (or faucet) might be labeled H, so his faucet might be labelled BAT-H.

Along with that, the definition bears reference to the clue as a whole, and may be properly expanded as:

$$\overbrace{\textit{where a superhero might label his faucet}}^{\texttt{definition}} \textit{ and } \overbrace{\textit{how a superhero might label his faucet}}^{\texttt{wordplay}}$$

This clue represents the upper level of challenge for a computer based solver, being unique structure, self referential, using very specialist knowledge and oblique humour.

Part IV
Optimizations

8 Algebraic + computational simplifications

8.1 Pruning out equivalent trees (Canonization)

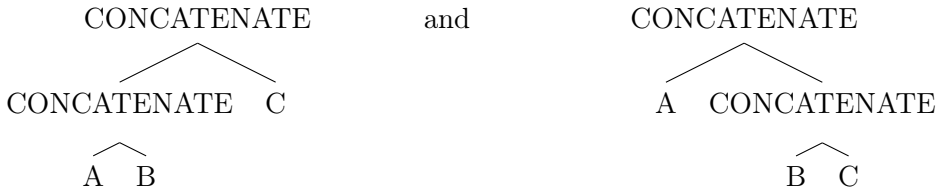
8.1.1 Motivation

One large factor in the rapid proliferation is in the our binary tree representation of concatenation. While keeping them in a similar representation to the rest of the expression nodes in

our naive solution kept their representation in a similar form to the rest of the nodes, the fact that no indicator is required to generate a concatenation node means that any expression of two ore more words can generate them.

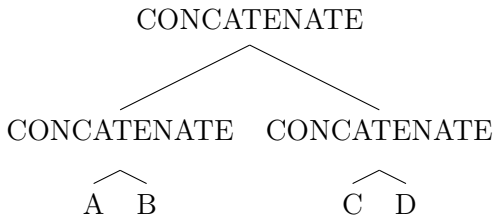
The number of trees with n leaves is given by the (n-1)$^{\text{th}}$ Catalan number[3], so ignoring any other type of expression (anagram, etc.), for a clue of length n we have C_{n-1} trees created with each of the clue words taken as a leaf (synonym) node. This number grows rapidly as the clue length increases, and yields an increasingly large number of parses.

Due to the associativity of concatenation, each of these parses evaluates to an identical output:

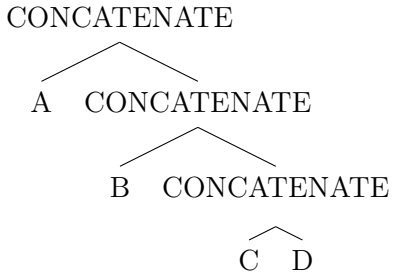


both yield the output `ABC`. This means that much of our outputted parses are identical and therefore redundant.

One strategy to deal with this would be to perform canonization on the trees, and prune all concatenation trees which don't conform to our decided 'ideal tree'. For example, we could choose to create a right-handed binary tree, wherein trees such as:



would become



The problem with this solution is that we need to look ahead while parsing: the above parse only is acceptable if the parse of `A` also doesn't produce a concatenation – this means we can't parse recursively as before.

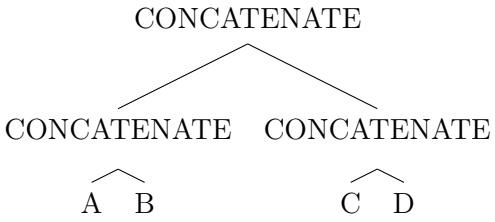
[3]Catalan numbers are given by the formula $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \geq 0.$

8.1.2 Implementation

We can instead define a new version of our Concatenation Nodes which, instead of describing a binary tree by storing the data as two parsetrees:

```
data ParseTree = ConcatNode ParseTree ParseTree | [...]
```

structured as:

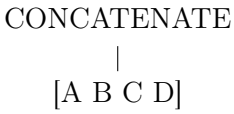



```
(as well as 4 other equivalent trees)
```

instead stores it as a forest, i.e. a list of trees –

```
data ParseTree = ConcatNode ParseForest | [...]
type ParseForest = [ParseTree]
```

structured as:



We define a new version of parseConcatNodes to reflect the new structure. This time, instead of considering all the ways to partition the wordplay of the clue into two parse, and subsequently combining each of the different parses of both of them, this time we need to consider all the ways to partition the string (which

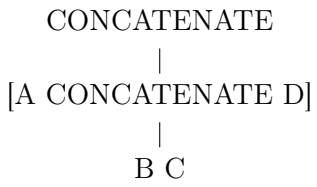
```
parseConcatNodes' :: String -> [ParseTree]
parseConcatNodes' xs n = let parts = partitions xs
       in [ConcatNode ys | part <- parts
                 , (length part) > 1
                 , ys <- [sequence . map parseClue $ part]  ]
```

the Prelude function `sequence`, which has the type `sequence :: Monad m => [m a] -> m [a]`, which when applied to a list of lists will provide all lists comprising of an element from each sublist:

```
sequence [ [1,2,3], [40,50], [666,777,888] ] =
  [ [1,40,666],[1,40,777],[1,40,888],[1,50,666],[1,50,777],
    [1,50,888],[2,40,666],[2,40,777],[2,40,888],[2,50,666],
    [2,50,777],[2,50,888],[3,40,666],[3,40,777],[3,40,888],
    [3,50,666],[3,50,777],[3,50,888] ]
```

8.1.3 Avoiding Nesting

Unfortunately, this solution alone will not prevent us from creating a forest of parse trees that itself contains a concatenation node:



leading to a even more parse trees than before!

In order to prevent our new concatenation nodes nesting again we need to define a version of `parseClue` which doesn't generate concatenation nodes:

```
parseClueNoConcat :: String -> [ParseTree]
parseClueNoConcat ys = let len = length . words $ ys in
  [Leaf ys]
  ++ (if len > 1 then parseConcatNodes ys else [] )
  ++ (if len > 1 then parseAnagramNodes ys else [] )
  ++ (if len > 1 then parseHiddenWordNodes ys else [])
  ++ (if len > 2 then parseInsertionNodes ys else [])
  ++ (if len > 2 then parseSubtractionNodes ys else [])
  ++ (if len > 1 then parseReversalNodes ys else [])
  ++ (if len > 1 then parseFirstLetterNodes ys else [])
  ++ (if len > 1 then parseLastLetterNodes ys else [])
  ++ (if len > 1 then parsePartialNodes ys else [])
```

and re-define our original `parseClue` as

```
parseClue :: String -> [ParseTree]
parseClue (Def def ys n) = let len = length . words $ ys in
  parseClueNoConcat ys
  ++ (if len > 1 then parseConcatNodes ys else [] )
```

8.1.4 Improvement Analysis

By cleaning up the redundancy in our different parses, we can improve our parsing function from exponential growth against clue length, to a low quadratic growth, as can be seen in **Figure 5** and **Figure 6**. As each parse may have thousands of solutions, this should represent a significant improvement in the number of outputs, and so the solve time, of each clue.

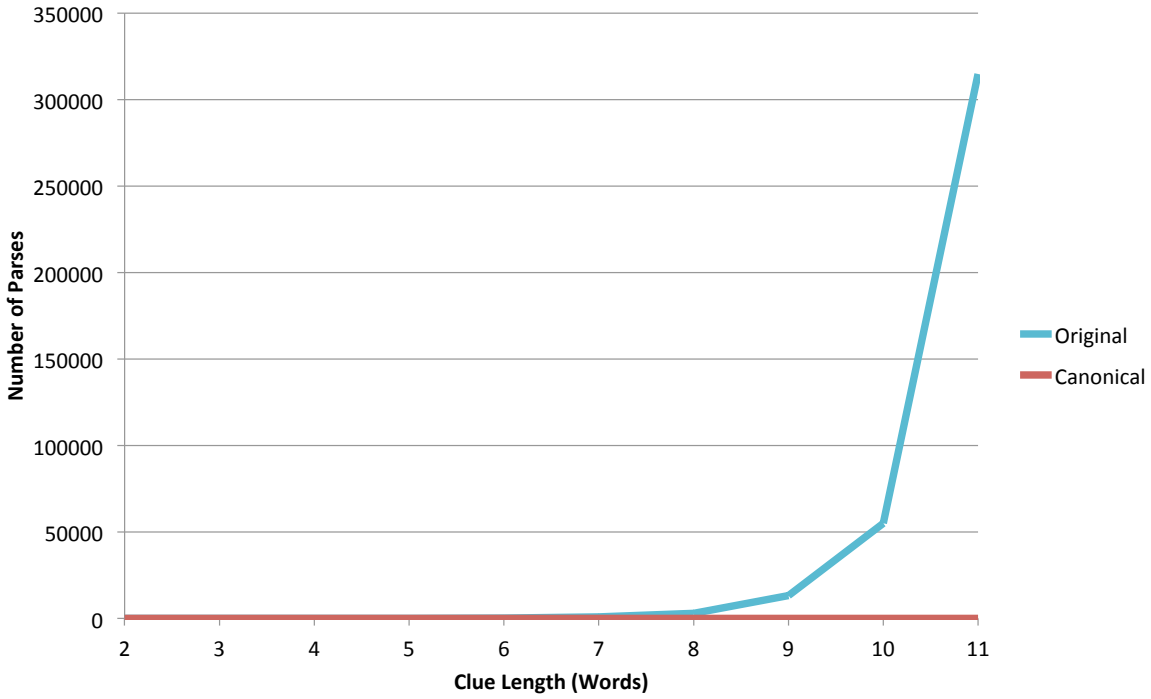


Figure 5: Average number of parses before and after canonization by clue length, averaged over 710 clues

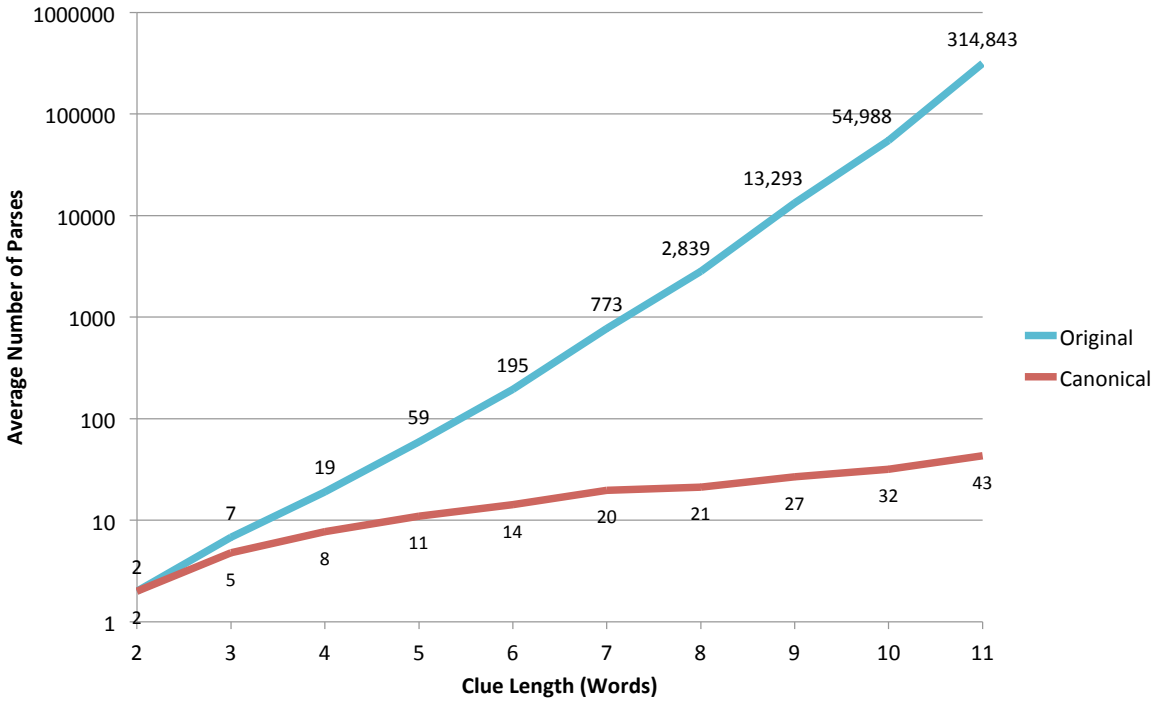


Figure 6: Average number of parses before and after canonization by clue length, averaged over 710 clues, on a logarithmic scale

9 Heuristics from Human Solvers

We can take cues for further improvements to our solving process by considering the heuristics that a human solver uses to navigate the huge state space and find the correct solution without having to check hundreds and thousands of possible solutions.

9.1 Filter parses by output length

9.1.1 Motivation

Here we seek to mimic the following thought processes of a human solver:

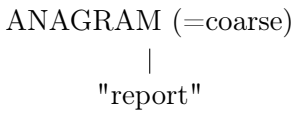
> *"This can't be an anagram of that word, as that'd only make 6 letters, and the clue is 9"*

> *"We can't have an insertion here, as we've already got 5 letters, and so if we add another 5 then it's too long"*

These are constraints on the parses that we can generate based on an understanding of the maximum and minimum number of letters than a given reading of a clue could produce. In the first example, a clue such as

```
Report coarse players (9)
```

could identify 'coarse' as a possible anagram indicator, and yield a parse such as

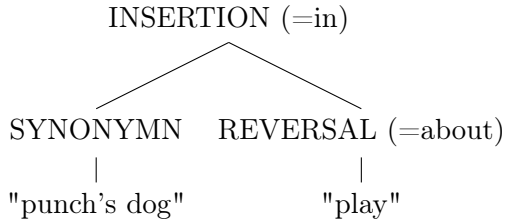


This, however, can never yield a solution that is 9 letters long, so a human solver, and so our improved computer solver, will not consider it for further evaluation.

In the second example, we see that we may also need to consider the parse recursively to calculate the total length parameters:

```
Punch's dog in play about bishop (4)
```

can be parsed to



Since the reversal of play ('yalp') is already 4 letters long, we can see that which ever word we choose to signify `punch's dog` will increase the length of the evaluated solution over the prescribed solution length of 4.

9.1.2 Implementation

We can recursively evaluate a parse to determine its maximum and minimum lengths, to check that the maximum is at least as big as the desired output length, and the minimum is at least as small.

We define the functions `minLength` and `maxLength` :

```
minLength :: ParseTree -> Int
minLength (ConcatNode trees) = (sum . map minLength) trees
minLength (Leaf string) = let x =
            minimum ( map length (string : syn string)) in x
minLength (AnagramNode ind strings) = (length . concat) strings
```

```
minLength (HiddenWordNode ind strings) = 2
minLength (InsertionNode ind tree1 tree2) = (minLength tree1)
                                            + (minLength tree2)
minLength (SubtractionNode ind tree1 tree2) = minimum[
                           (minLength tree2) - (maxLength tree1),1]
-- and definitions for other clue types
```

Some clue types are can be defined directly from their inputs – both the maxium and minium length of an anagram node is the lenght of the input string – while an insertion node need to be defined based on the maxium and minimum of the two subtrees.

Notable is that here we see some 'contextual bleed' from evaluation across into the parsing, as we consider the semantics of what the thesaurus could yield for a Leaf synonym node in determining its minimum length.

It's also worth noting that sometimes we need to make a judgement: what is the minimum that a Hidden Word could yield?

From these definitions, and similar ones for maxLength, we can check a parse for validity.

```
valid_parse_length :: Parse -> Bool
valid_parse_length (Def d clue n) = (minLength clue <= n)
                                  && (maxLength clue >= n)
```

and so redefine parse as

```
parse = filter valid_parse_Length . concatMap parseClue
```

9.1.3 Analysis

Figure 7 shows the effect on number of parses generated following the addition of the parse length constraints.

This filtering constraint now means that many clues now yield 0 parses. Some of these are clues that could never be correctly parsed, while some are clues which we can generate correct parses, but do not have the thesaurus and synonym data to solve the clue.

This transformation is, though, safe – any parse that previously would have generated the correct answer will not be filtered out.

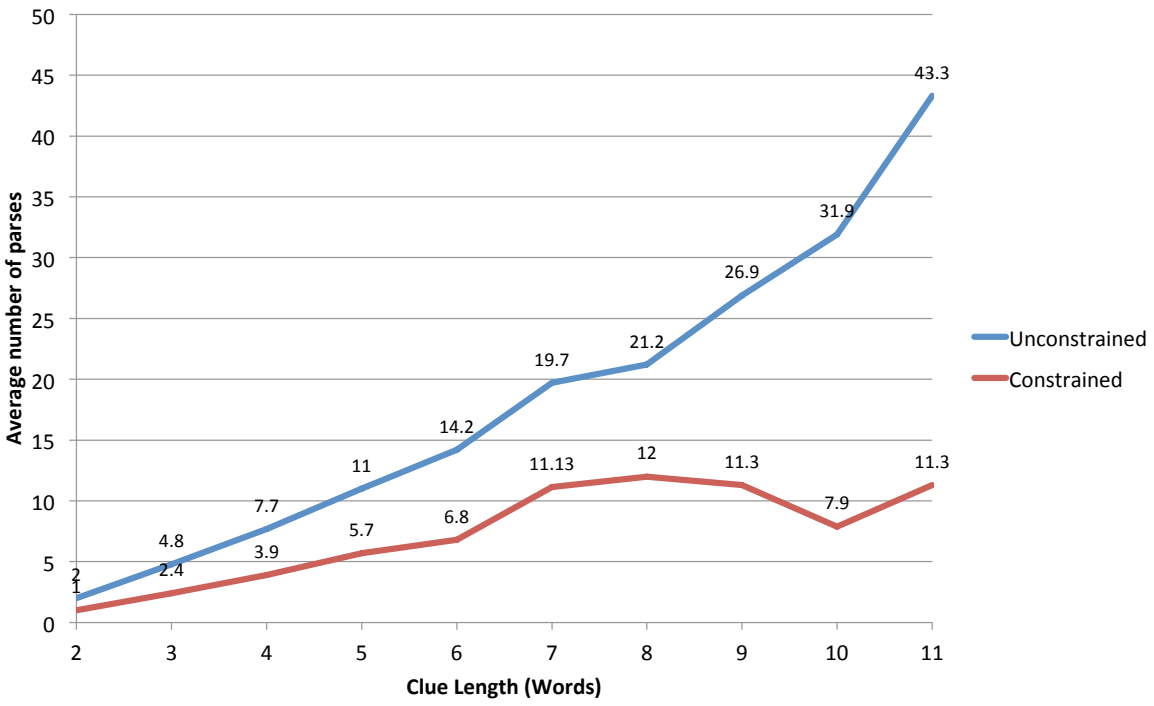


Figure 7: Average number of parses before and after canonization by clue length, averaged over 710 clues, on a logarithmic scale

9.2 Taking Advantage of Lazy Evaluation

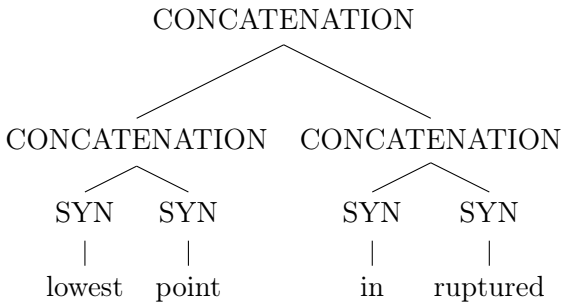
9.2.1 Motivation

Although we are now generating far fewer parses, we still have some solveable clues generating 100+ parses. This means that for these clues we will have to perform on average $n/2$ evaluations – lazy evaluation means that our use of the `head` with `filter`will yield the first result computed from the head of the list toward the tail.

We could take further advantage of the intuition that some parses are more likely, given the input words. For instance in the clue `Lowest point in ruptured drain (5)`, we see the angram indicator 'ruptured' next to a 5 letter word:

$$\begin{array}{c} \text{ANAGRAM (=RUPTURED)} \\ | \\ \text{drain} \end{array}$$

with the definition `lowest point`, intuitively feels more likely than



cluing the definition`drain`.

More formally, we're looking for a heuristic which weights toward consuming words into indicators for more 'interesting' clue types: in that clues using expressions more varied than synonym and concatenation are considered better clues, and so are more likely than not if they are an available parse.

Furthermore, these expressions consume more of the string in indicators than other types (reversal nodes consume one word from the clue as its indicator, while synonyms and concatenation both don't consume any indicators) and are less likely to produce nested parse trees (both anagrams and hidden word nodes treat their input as a pure string to be transformed, and so do not generate any nested parse trees). This means that clues featuring these types tend to be less complex.

Both of these factors make them good candidates to evaluate sooner than other options.

9.2.2 Implementation

We define a method `cost` which gives a weighting to a given ParseTree

```
cost :: ParseTree -> Int
cost (ConcatNode trees) = 2 * (length trees) + sum (map cost trees)
cost (AnagramNode ind strings) = 1
cost (HiddenWordNode ind strings) = 4
cost (InsertionNode ind tree1 tree2) = 4 + cost tree1 + cost tree2
cost (SubtractionNode ind tree1 tree2) = 3 + cost tree1 + cost tree2
cost (ReversalNode ind tree) = 2 + cost tree
cost (Leaf string) = 8 * length (words string)
cost (FirstLetterNode ind strings) = 2
cost (LastLetterNode ind strings) = 2
cost (PartialNode ind tree) = 6 + cost tree
```

we can then define

```
cost_parse :: Parse -> Int
```

```
cost_parse (DefNode s tree n) = cost tree * (length_penalty s)

length_penalty :: String -> Int
length_penalty ws = 60 + (length (words ws))
```

which can then be integrated into our definition of `parse`:

```
parse = sortBy cost_parse . filter valid_parse_length . concatMap parseClue
```

It should be noted that the weights here are intuitive only.

Leaf node has a high weighting against consuming long lists of words – this is to prevent them from being low scoring (as they consume large portions the clue) while being unlikely to yield the correct answer.

9.2.3 Analysis

The weighting above mean that the correct parse had the highest score in 70% of the clues that the system can solve, as opposed to approximately 10% when not sorted by weight. In cases where the clue can not be solved, the order of the parses is irrelevant.

Generate much more data for this and display in a nice way.

9.2.4 Determining a correct weighting

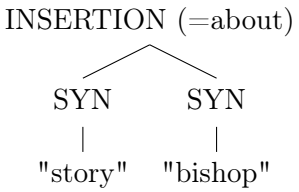
While the current weighting given has been developed though trial and error to be reasonably successful, a more structured approach to detemining the correct weighting could generate even better results. Using a large dataset of clues and the correct parses, hill climbing or statistical analysis of clue types could produce optimal numbers.

9.3 Constrain length while evaluating

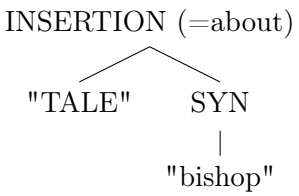
9.3.1 Motivation

While evaluating a parse tree of a given clue, we expect the overall length of the generated solution to be equal to the length specified in the clue. Furthermore, while evaluating different sub-trees of a given parse tree – either different branches of a concatenation list, or the two constituent parts of an insertion or subtraction experession – the solutions generated from one influence and limit what can be generated from the others.

For example, in the clue `Story about bishop and food (5)`, if we are evaluating the parse



then the partial evaluation of the left hand branch to one of its possible evaluations



means that as the subsequent evalaution of `SYN “bishop”` should only yield solutions that are one-letter one, in order to stay within the constraint of a five-letter solution. If we can successfully apply this constraint, then we can limit the subsequent evaluations of this partial parse tree to one or two, rather than the order of 100.

9.3.2 Implementation

In the example above, we see the length constraints preventing overflow - that is, a maximum length which the generated solution should not exceed. We also need to constrain against 'underflow', wherein the evaluation fails to yield enough letters to fit the solution. Constraining both maximum and minimum length will have the effect of forcing the generated solution length to be equal to the prescribed length.

We can therefore define a datatype to carry both of these constraints.

```
newtype Constraints = Constraints MaxLength MinLength
```

In some cases, we will not be able to prescribe a definite maximum length for a clue: in the case of a subtraction expression of parse trees A and B, where the evaluation of tree A will have the evaluation of B removed from it to yield the final solution, the length of clue A will exceed the overall solution length by an amount only limited by the length of B.

We also, then define `MaxLength` and `MinLength` as new datatypes.

```
data MaxLength = Max Int | NoMax
data MinLength = Min Int | NoMin
```

Although we could use the `Maybe` monad here, by defining our own datatype we can subsequently take advantage of Haskell's type class system later on to allow us to treat these, and other constraints, in a similar way.