Solving Cryptic Crosswords through Functional Programming

Michael Skelly
Department of Computer Science
Imperial College

April 25th 2014

Part I
Introduction to the Problem / Field

Cryptic crosswords are widely thought to be at the crossroads of various fields of human endevour considered to be right at the limit of current AI and Machine Learning – featuring wit, slang, allusion, linguistic ambiguity and generally deliberate trickery. Along with this, they possess other characteristics that make brute force solutions difficult, if not impossible: the state space of all possible crossword grids is of the order 10^{90}(compare, for example, an upper bound on all the possible chess positions is merely 10^{50}), and worse still, a solution to a grid is non-trivial to verify (as the verification process is the nearly same as solving the clue!)

Nevertheless, techniques from combinatorics, compiler design and NLP and AI all have applications that can help elucidate and simplify the problem, along with heuristics adapted from both human solvers and analytical optimisations that can help improve the time taken to arrive at at the correct solution.

I have developed a system which can solve a significat percentage of cryptic crossword clues

> "A provost at Eton once boasted that he could do The Times crossword in the time it took his morning egg to boil, prompting one wag to suggest that the school may have been Eton but the egg almost certainly wasn't." – Citation Needed

Part II

Literature Review

1 Summary of Cryptic Crosswords

1.1 Definition of Cryptic Crosswords

To start, let us provide some basic definitions around crosswords and their taxonomies.

A crossword is a puzzle, usually published in newspapers or magazines. They consist of a grid of squares, often 15 x 15. Some of the squares are white (i.e. blank) and some are blacked out. Any contiguous run of more than one white square, either down (vertically) or across (horizontally, left to right) is a space for a word, to be written. These are marked by numbers in the initial square (the top-leftmost one), and referred to by those numbers, and the direction (e.g. '5 down', '8 across'). Horizontal runs can overlap vertical runs, and at the points at which they do, each of the two words, when written in, must have the same letter in that square. Along with the grid are a set of clues, which the solver can use to determine which word to write in each space (the 'answer' or 'solution'). The aim of the puzzle is to find the set of solution words such that each clue's solution is correct for that clue, and fits in the grid correctly, with respect to the overlapping words.

Grids can be very densely white, with few black squares and most squares shared by two words (usually called AMERICAN STYLE) or more sparse, with fewer overlapped clues (called BRITISH STYLE). Clues can also be in two styles. STRAIGHT or QUICK crossword clues usually provide a single straightforward indicator as to what the correct word might be - often a synonym for the clue ('Joyful' = 'Happy') or a missing word ("Stitch in _ _ _ _ saves nine" = "Time"). CRYPTIC clues are less straight-forward, appearing on the surface to be a valid syntactic utterance in English, but actually consisting of a definition (as in the Straight clue) and some wordplay which the solver can use to arrive at the same answer as with the definition by apply a series of transformations and operations. The challenge is that the definition and the wordplay are not clearly separated, and that there are multiple ways to apply to the transformations, but with only on yielding the correct answer.

It is the task of determining the correct answer for this type of clue that this report will address.

1.2 Cryptic Crosswords in the Literature

While not a topic well covered in scientific literature in general, what few analytical studies around cryptic crosswords there are tend to be classifiable into three main groups

1.2.1 Generation of Cryptic Clues

The largest body of work that exists is centered around the generation of cryptic clues, focused largely around analysis of how string literals from a predetermined answer can be transformed by set clueing patterns, as well as some work around measures of the quality of generated clues.

1.2.2 Interpreting Clues

The next set are the select few who have done prior, similar investigations into interpreting cryptic clues, with some work put into formalizing definitions and notation for the sorts of clue types that appear in the majority of cryptic crosswords, and some attempts at solving based on these interpretations.

1.2.3 Other Work

There has also been some work done towards solving non-cryptic crosswords probabilistically, working on whole-grid solutions rather than individual clues. There are also some more left-of-field studies done: statistical studies into errors made during manual solving, and psychological studies into solving.

1.3 Complexity

A variety of factors make solving cryptic crosswords a difficult problem:

Ambiguity Cryptic crosswords are deliberately ambiguous. Instruction indicators are indistinguishable from string literals, which are identical to words' semantic meanings. Often, the setter will deliberately chose words to give rise to further ambiguities. For example, the *Telegraph* printed

```
Bug starts to move in dark, glowing endlessly (5)
```

cluing for 'MIDGE'. Usually "endlessly" and similar mean "remove the last letter", but here it is one of five consecutive words to form an acronym from, with the word "starts" as an indicator.

State Space Even with only a few different clue types, the number of different readings of one clue based on those grows exponentially with the length of the clue. This means that unless heuristics are applied, the evaluation time for a whole grid longer clues may be unfeasibly long. Even longer than it takes me to do the Times Crossword.

Lack of Standardization Although all cryptic crossword share some common conventions, there are no fixed rules shared between publications for what can and can't be a clue, indicator etc.. Although most publications have internal guidelines or style-guides, these are not accessible to the solver, and some publications (such as the *Guardian*) have named setters whose styles and self-imposed rulesets differ, even between one publication. Alistair Ferguson Ricthie, who set for *Listener* for many years, referenced the concept of fairness in his book *Armchair Crosswords* in 1946. He defers the judgement of fairness to a notional rulebook:

> We must expect the composer to play tricks, but we shall insist that he play fair. *The Book of the Crossword* lays this injunction upon him: "You need not mean what you say, but you must say what you mean." This is a superior way of saying that he can't have it both ways. He may attempt to mislead by employing a form of words which can be taken in more than one way, and it is your fault if you take it the wrong way, but it is his fault if you can't logically take it the right way.

Although *The Book of the Crossword* there have been many books written on the subject of what should and should not constitute a valid cryptic crossword clue. One of the most notable and influential was written by *Observer* setter Derrick Somerset Macnutt, both cluing and writing under the name Ximenes, in his book *Ximenes on the Art of the Crossword Puzzle*. The book contains many in-depth guidelines about what a fair clue entails, summed up by his successor Azed (Jonathan Crowther, born 1942):

> A good cryptic clue contains three elements:
>
> 1. a precise definition
> 2. a fair subsidiary indication
> 3. nothing else

A crossword setter following these rules is said to adhere to 'Ximenean principles' and their produced work to be Ximenean. Most mainstream crosswords exist on a continuum between being more closely Ximenean (examples include *The Times*, the *Independent*) to being very libertarian (e.g. *Guardian*). No crossword in a major UK newspaper is 'strictly Ximenean'.

Knowledge Base As well as being made up of encrypted and hidden meanings, cryptic crosswords also draws on a diverse knowledge base of synonyms, abbreviations, facts etc. These can include information as diverse as names of capital cities, common sayings, and the fact that one may carry a wallet in ones pocket.

In order to run a fully working cryptic crossword solver against any arbitrary clue, all of these pieces of information must be encoded, stored and accessible to the solver in a machine readable form. Understandably, this is subject to an entire field of study itself.

1.4 Programming Language Analogues

Much of the current work on interpreting crosswords draws on work by Backus, Naur and Chomsky in creating a specification for the grammar of crosswords. While these frameworks are useful for describing many different languages, interpretations of the grammar of cryptic crosswords seem to be perversely somewhat closer to mathematical and programming languages than to natural language. In some ways, the cryptic clue as a whole can be thought of as a program that generates the output string as its answer. The wordplay section is analogous to a program, and the definition section of a clue could be thought of as a checksum to verify the final answer.

1.4.1 Lexing, Parsing, Evaluating

The steps for compiling and running a computer program apply also to solving (or 'running') a crossword as a program. Each word in the input string needs to be tokenized, parsed into a relevant structure and then that structure evaluated to produce the final answer. Unusually for a programming language, however, the grammar of a cryptic crossword is highly ambiguous, and requires complex parsing. Firstly, programming language are only usually required to output the one valid abstract syntax tree, however here we may need to output many thousands in order to evaluate them to see which yields the correct answer. Furthermore, the grammar cannot be expressed without using complex context-sensitive features such as lookbacks, lookaheads and backtracking. Most major programming languages are parsed without these features, allowing information to flow in one direction from the lexer to the parser. To parse a cryptic crossword, lexing and parsing need to take place simultaneously in a process referred to as called "Scannerless Parsing".

1.5 There's No Accounting for Wit

Along with clearly defined and program-like cryptic crossword clues, there exist other clues that rely on humour, imagery and wit, rather than following the regimented classical structure, as set out by Ximenes. Some examples include:

```
Flower of London? (6)
    (= THAMES, flower = that which flows)
In which you can get three couples together and have sex (5)
    (= LATIN, 'sex' is 6 in Latin)
```

Clues such as these, and the question of computerised wit and humour, unfortunately exist out of the scope of this project.

2 Parsing Frameworks and Notation

Some different notations for denoting parsing of cryptic clues have come out of previous work – in order to properly provide a rigorous analysis of the structures and conventions of cryptic crosswords, it is necessary to analyze and choose a framework in which to do it.

2.1 LACROSS

William and Woodhead produced language called LACROSS, which forms a sort of calculus for describing crossword clues. They also provide a BNF definition of this grammar. Their clues are of the general form

```
Clue := Δ = G | G = Δ
```

the orientation of which corresponds to the order in which we find the definition (Δ) and the wordplay (G) in the clue. The wordplay may be further expanded out – the wordplay section of the clue is expressed as a sequential annotation for the constituent parts, either as ‘text’ (t), ‘shortening’ (S) (etc.) or as placeholders for the operators (*), which are detailed afterwards, including a reference to the substituted indicator. So for instance:

```
Get in odd bit of colour (5) [= tinge]
t* = Δ, a (odd, a)
```

There are several issues with this grammar. Firstly, all unitary operators are treated the same, as are all binary operators, and there is some issue with binding and precedence which they address with an underlining notation, in addition to brackets. Secondly, the grammar attempts to include both the structure of the parsed cluing and how that structure relates to the original sequence of words at the same time. As a result, we end up with complex grammar that does not aid human parsing of the solution well, nor does it lend itself easily to computer or mathematical manipulation

Still, they have provided the basis for future work, and begun a basic enumeration of clue types.

2.2 Simple Clue Markup Language

Proposed by Hall and Rapanotti, Simple Clue Markup Language (SCML) attempts to notate the structure of the solution directly onto the clue.

Double underlining is used to denote the definition, underlining denotes an operator, with its class as an optional subscript, with scope provided by brackets and concatenation (and definition/wordplay separation) given by a semi-colon. Thus in their given example:

```
Note the shuddering appliance Bill regularly installed, noisy thing (6,7)
```

$$\underline{\texttt{Note}}\texttt{;(the)}\underline{\texttt{shuddering}}_{\mathrm{a}}\texttt{;(appliance,(Bill)}\underline{\texttt{regularly}}_{\mathrm{t}}\texttt{)}\underline{\texttt{installed}}_{\mathrm{e}}\texttt{;}\underline{\underline{\texttt{noisy thing}}}$$

Note’ often indicates a musical note, resolving to one of ‘a’ to ‘g’, ‘do’, ‘re’, ‘mi’, etc;

> ‘the shuddering’ may be an anagram indicator applied to ‘the’;
>
> the ‘regularly’ of ‘Bill regularly’ may indicate alternate letters (‘t’); i.e., ‘bl’ or ‘il’; and
>
> ‘installed’ suggests the embedding (‘e’) of those letters within something meaning ‘ appliance’.

In this, we have no markup differentiation for literal strings (‘Bill’) against words with their semantic context (‘appliance’), and we also take certain words that reduce to abbreviations (‘Note’) to be non-deterministic nullary operators. With some changes and additions (tagging of string vs. semantic word, for example), this markup serves as a good way to represent a parsing of a clue in a human readable way. It even has the advantage that a printed clue could be annotated (carefully) by hand, as a teaching aid, for example. Unfortunately, the language as it stands is not expressible as a BNF grammar, nor is it a particularly good format for representing the clue and its parsings internally in a program (as it would need to be re-parsed to use!)

2.3 Clue-answer notation

There are several emergent solutions within online cryptic crossword communities for notation to explain solutions derived from clues. From http://cryptics.wikia.com:

> Consider the down clue A message from the setter, hauled up with broken arm after heroin withdrawal (8) yielding the answer TELEGRAM. The corresponding wordplay, having the prolix and possibly ambiguous explanation THE next to LEG reversed next to an anagram of ARM, all with H (heroin) removed could be concisely represented in clue-answer notation simply as T[h]E,GEL<=,(ARM)*.

These meanings are not fixed, but some definitions are given here:

ABC<= or ABC (rev.) ABC reversed. The (rev.) notation is most commonly used when the wordplay consists of a single reversal.

[abc] or -abc or (abc) Letters abc removed, as in[c]OUNT to represent ’count’ with c removed; the convention is to use lower case for the removed letters.

(ABC) Letters placed inside others, as inC(AND)ID to mean ’and’ inside ’cid’.

"ABC" Homophone of ABC.

(ABC)* Anagram of ABC.

A+B or A,B A concatenated with B. Sometimes both notations are used together where ambiguities may arise.

aBcDeF Alternate letters of ABCDEF (shorthand for[a]B[c]D[e]F).

2.4 PICCUP

Hart and Davies define what is currently the most satisfying proposal for a formal syntactical definition of cryptic crossword syntax, in a loosely BNF grammar. Theirs is the only current definition that closely resembles a usable formally defined language.

Their interpretation only specifies the grammar in terms of building an abstract syntax tree, rather than attempting to include a notation for clue or answer.

```
Anagram → Synonym(.Equ Indicator).AnagramSentence
/AnagramSentence(.Equ Indicator).Synonym
AnagramSentence → AnagramPointer.AnagramMaterial
/ Anagram Material.Anagram Pointer AnagramPointer~ Word(.Word)*
AnagramMaterial → Word(.Word)*
Synonym → Word(.Word)*
Equ Indicator → Word (.Word)*
```

2.5 Syntactic and Metasyntactic Conventions

Here we apply a similar convention to Hart, in using a modified Backus Naur Form (BNF). We will later see that a context-free grammar may not be sufficient to model a cryptic crossword, and may have further deficiencies as a basis for finding a solution. Nevertheless, we will adopt a similar notation:

```
→ = is composed of
, = followed by
| = or
(x) = x is optional
x* = 1 or more occurrences of x
(x)* = 0 or more occurrences of x
```

We also take the BNF conventions

```
Word = non-terminal symbol
“word” = string literal
[x, y, z] = list containing x y and z
(x, y) = pair x and y
```

For clarity, we additionally define:

```
String = [any string literal]
```

2.6 Context Free?

The grammar described in this paper not a regular grammar (for example: any of the binary operators generate two non-terminals), but it can be formulated as a context-free grammar.

We define a CONTEXT-FREE GRAMMAR (CFG) as one in which the expansion of a non-terminal is not affected by the symbols before and after it.

While we can certainly define a working grammar for cryptic crossword clues in terms of a CFG, it may be useful to consider other options as a means of reducing the number of trees generated during the parsing phase to speed up the evaluation phase. We could take, for example, the clue length as a contextual variable: in that case, a 6 letter clue whose parse tree contains an anagram of a 4-letter string cannot yield another anagram of 5 letters.

```
             Cons Node
    _______________|_______________
Anagram                Wordplay
  |
String
  |
``apples''
```

In this example, the wordplay on the right should not be expanded out to an anagram node featuring a string of 5 characters (to consume, say, the string "mixed pears").

2.7 Syntax vs Semantics

Due to the ambiguous and duplicitous nature of the structure of cryptic crosswords, especially the deliberate challenges in the lexing phase, unclarity between the boundaries between parsing the syntax and evaluating the semantics emerge.

Strings consisting of one or more words can be at once tokens representing different operators, they can be strings, and can be split in multiple ways into combinations. This is especially true when we have token that, in the original text, represent their semantic meaning in English, and evaluate out to a finite number of equivalent words (roughly, synonyms: see later for discussion about this equivalence relation).

Hall and Rapanotti treated these roughly as their own operators: so the string "rough" would parse to the token Rough, which later evaluates to a finite number of definitions. This may, indeed, be tempting if we had a limited number of candidate words. And, indeed, we do need to differentiate these from raw string literals that are subject to Hidden Word or Initial Letter operators, or occasionally concatenated in their raw form (for example the string "it" is sometimes taken as given where necessary) .

I think a more manageable way and satisfying way to consider these options is to consider them subject to an invisible 'word' operator. This keeps the semantics and syntax more separate, but certainly poses some challenges for a parser / lexer.

3 The Cryptic Crossword Clue

3.1 Structure of a cryptic clue

A cryptic crossword differs from a normal crossword in that the clue for each answer consists of two parts. The first is the definition, which performs the same function as a clue in a 'regular' crossword. The answer to the clue is usually a synonym for the definition ('circular' and 'round') or may be an example of the definition ('farm animal' and 'pig'). Other forms that the definition may take will be discussed later on. The second part of the clue is the wordplay. This is an encoded and often ambiguous second method of deriving the answer, using techniques such as anagram, substitution and concatenation. The clue as a whole is presented as a concatenation of the two parts, sometimes with a subsidiary word indicating that one can be derived from the other (for example, 'from' or 'is'). We can present this breakdown as:

```
Clue → Definition, (Indicator), Wordplay
     | Wordplay, (Indicator), Definition
```

The final clue will often resemble a valid English utterance, although this 'surface reading' (i.e. {clue}) very rarely has any relation to the answer. Later on we will consider other information and context within the definition of a clue.

3.2 Definition

The definition of the clue consists of one or more English words. The answer to the clue will be a word or phrase that fits an appropriate equivalence function (that we will define later).

The definition carries a variety of linguistic features with it that the overall answer, and so the answer as derived by the wordplay, must match. These include aspect (noun, verb, adjective), plurality (tree, trees), tense (go, going, gone). These features may also be considered as 'context' to the clue itself.

3.2.1 Formally

We can define the definition as

```
Definition → Words
```

3.3 Wordplay

The wordplay section of a clue is a set of deliberately ambiguous instructions that allows the solver to arrive at the eventual answer. As the instructions are ambiguous, multiple possible parsings of the instructions are possible. Some of these parsing will not lead to a valid English word:

```
Imbecile, bonkers, in a cult (7)
==> Wordplay ‘Imbecile, bonkers = definition ‘in a cult’
==> Anagram ‘imbecile’ [indicator = bonkers] = definition ‘in a cult’
==> ??? (no anagrams of imbecile in english language)
(correct reading was anagram of in a cult = lunatic)
```

Others will lead to a valid English word, but one that is not equivalent to the definition:

```
Minder shredded corset (6)
==> Wordplay ‘minder shredded’ = definition ‘corset’
==> Anagram ‘minder’ [indicator = shredded] = definition ‘corset’
==> ‘remind’ = definition ‘corset?’ X
(correct reading was anagram ‘corset’ = escort = minder)
```

The solver must find the correct parsing of the wordplay that yields the correct definition: even though they may not know which part is wordplay and which is definition.

3.4 Special Operators

I include these two operators first, as they really form the backbone or basis of other clues. They are also unique in being implicitly clued, rather than requiring an indicator word to signify their presence.

Word Equivalence In the most simple of clues, we have the definition, along with a word or phrase that is somehow semantically equivalent to that definition. [1] A clue that contains just this structure is said to be 'double definition'

```
Metal guide (4) [= LEAD]
```

However, even in this simple example we see that this equivalence relationship is not at all straightforward. While 'guide' and 'lead' are synonyms (as verbs in the present tense), it's not true that 'lead' is a synonym for 'metal'. We must also include 'for example' in this relationship too, which causes us to have to discard reflexivity. Although 'metal' can be a clue for 'lead', it's not the case that 'lead' can be a clue for 'metal' (in that case, we signify 'an example of' by writing 'lead, say' or 'bronze, for instance').

We also include abbreviations, which are perhaps more closely related to synonyms, although not usually found in thesauruses, along with some useful 'setters favourites', where an abbreviation of a synonym or of an example is particularly useful for cluing a difficult letter combination used in a wordplay ('Books' becomes 'NT', for 'New Testament').

[1] In this case, it becomes a difficult task to be precise about exactly which of these is the definition and which is the wordplay! Sometimes there is a defined answer: From 'Oinking tendency? (8)' we get both 'pen chant' and 'penchant', and we can see from the letters required (no space) that the second half is the solution. In other cases, this may not be defined at all!

```
Words → Synonym | Abbreviation | Example
Synonym → String
Abbreviation → String
Example → String
```

The semantic task of evaluating this will be discussed later.

Concatenation While not strictly necessary for this grammar (as we have included a concatenation in our metasemantics, we could define multiple definitions of each operator in the form Operator → Indicator, Wordplay, (Wordplay)*), it makes sense to add this explicitly as it mirrors the structure of an explanation of a computer solution (i.e. the parse tree).

```
Concatenation → Wordplay (ConcatIndicator) Wordplay
```

This represents a key tool for cluers to create more complex wordplay clues in the form of a charade, where two or more parts can be split out (sometimes syllabically as in 'bath', 'tub', or sometimes otherwise 'bat','htub') and clued separately, and then later joined to form the overall solution.

3.5 Other Wordplay Operators

For the other wordplay operators, we define them in terms of our grammar, as well as discussing their semantic meaning.

```
Wordplay → Words | Concatenation | Anagram | Reversion | Contraction
         | Selection | Hidden Word | Containment | Subtraction
         | Homophone
```

These operators all include an indicator word to show they are being applied (as is far more common with operators in programming language parsing!) Each operator will usually have many different indicators (lists of anagram indicators on the web span multiple hundreds). Only select ones are included in the specification here.

3.5.1 Unitary Operators

Anagram A very commonly used operator in crossword clues is an anagram. These take the form of an indicator word that denotes that the anagram function is being used (called an 'anagrind' within cruciverbalist circles), along with the candidate letters to be anagrammed. The simplest form of this gets the candidate letters verbatim from the clue:

```
Anagram → Anagrind, String | String, Anagrind
```

Sometimes, however sometimes there is some sort of operation applied to the letters before the anagram is applied. For example:

```
Comic bare for short comedy play (7,5)
==> Wordplay ‘Comic bare for short comedy’ = Definition ‘play’
==> Anagram ‘bare for short comedy’ [anagrind = ‘comic’]
==> Anagram (‘‘bare for’’ + Shorten ‘comedy’)
==> Anagram (‘‘bare fore’’ + ‘‘comed’’)
==> Anagram (‘‘bare fore’’ + ‘‘comed’’)
==> Anagram (‘‘bareforecomed’’)
==> ‘‘Bedroom Farce’’
```

In which case we find the more general case one proposed structure:

```
Anagram → Anagrind, Wordplay | Wordplay, Anagrind
```

Wherein we know that the repeated evaluation of the Wordplay will eventually result in a string literal that can be anagrammed. In *Art of the Crossword Puzzle,* Ximenes argued against this form of indirect anagram:

> Secondly – and here, for once, I differ from Afrit – I hate what I call an indirect anagram. By that I mean "Tough form of monster" for HARDY (anagram of HYDRA). There may not be many monsters in five letters; but all the same I think the clue-writer is being mean and withholding information which the solver can reasonably demand. Why should he have to solve something before he can begin to use part of a clue? He has first to find "hydra" – and why shouldn't it be "giant"? – and then use the anagrammatic information to help him think of "hardy". ... My real point is that the secondary part of the clue – other than the definition – is meant to help the solver. The indirect anagram, unless there are virtually no alternatives, hardly ever does. He only sees it after he has got his answer by other means.

Even so, most setters that claim to be Ximenean will allow small abbreviations and contractions (to be defined later) to be included in their clues. We therefore must define a new class which includes String Literals as well as the abbreviation where appropriate.

```
Anagram → Anagrind, StringWordplay* | StringWordplay*, Anagrind
Anagrind → ‘‘free’’ | ‘‘novel’’ | ‘‘comic’’ [...]
StringWordPlay → String | Abbreviation | Contraction
```

Reversion Clues can also be reversed. While this is functionally a subset of anagrams, there are some crucial differences. Firstly the ‘directionality’ of the clue (i.e. whether it is a ‘down’ or an ‘across’) comes into effect, in determining the sorts of indicators that can form it: “turned back” may only apply to ‘across’ clues, where “taken up” may only apply to ‘down’ clues. Further, these clues are usually taken to be ‘fairer’ game for subsequent operations to be applied to the target of the reversion. Therefore, a clue with nested wordplay such as

("Stressed, made upside-down pudding (7)" = DESSERT) would be acceptable, where an equivalent clue as an anagram ("Stressed, cooked up pudding") would often not be seen as Ximenean.

```
Anagram → ReversionIndictator, Wordplay | Wordplay, ReversionIndictator
ReversionIndicator → “around” | “turned back” | “taken up” [...]
```

Contraction Clues of this form range from specific, such of first/last letters ('first in line' = 'l', 'last of the Mohicans' = 'm') to more general operators ('mostly harmless' can yield 'armless', 'harmles', 'harmle'...) whose definitions are more flexible.

```
Contraction → FirstLetterContraction | LastLetterContraction | GeneralContraction
FirstLetterContraction → PreFLCIndicator, Wordplay | Wordplay, PostFLCIndicator
LastLetterContraction → PreLLCIndicator, Wordplay | Wordplay, PostLLCIndicator
GeneralContraction → PreGCIndicator, Wordplay | Wordplay, PostGCIndicator
```

Selection There are three similar operators here: A pair which select even or odd letters respectively, and one which takes initial letters across multiple words. These are rarely, if ever, applied to anything other than pure strings. The initials indicator needs to be applied to an argument consisting of multiple words.

```
Selection → Evens | Odds | Initials
Evens → EvensIndicator, String | String, EvensIndicator
Odds → OddsIndicator, String | String, OddsIndicator
Initials → InitialsIndicator, String, ‘‘ ‘‘, String* |
           String, ‘‘ ‘‘, String*, InitialsIndicator
```

Hidden word The hidden word clue finds a word which appears as a substring (ignoring spaces) inside its operand. These typically only occur once per puzzle, and are always accompanied by a clear indicator. In this example clue:

```
'Smack which appears in East Anglian ports.(4)'
```

the solution to this example is 'TANG', (meaning 'smack' in the sense of 'taste'), and which is concealed (indicated by 'which appears') in 'easT ANGlian ports'.

```
HiddenWord → HWIndictator String | String HWIndicator
```

Homophone

Also called 'sounds like', this operator produces homophones of a given word. They may be spelled differently ('right' and 'rite') or the same but said differently ('Polish' and 'polish'). This operator is not applied to words that are both spelled and said the same, but with different meanings ('must' as an imperative and 'must' as a noun).

Often, if clues are straightforward, placement of this operator can determine the spelling of the answer.

```
We hear twins shave (4)
```

yields 'pare' whereas

```
Twins shave, we hear (4)
```

yields 'pair'. A formulation with the indicator in the middle, in this case, would result in a strong ambiguity. The homophone indicator is only applied to equivalence words, not to clued wordplay.

```
Homophone → HomophoneIndicator Words | Words HomophoneIndicator
```

3.5.2 Binary Operators

As with the unitary operator, each of the arguments of binary operators can be one or more words.

Containment Here are two styles of wordplay which are clued very differently, but are actually the same operator, which places one set of letter inside another. This is either presented as a insertion ('end inside ls') or as a containment ('ls around end'). This operation always preserves letter order, unless some nested indicator allows otherwise.

```
Containment → Wordplay ContainmentIndicator Wordplay
ContainmentIndicator → “inside” | “around” [...]
```

Subtraction In a subtraction clue, a number of letters are removed from the target. Usually, the target is some wordplay itself, although sometimes just a string literal. The letters to be subtracted are also often the product of some sort of cluing, although this is usually fairly limited in scope (abbreviations, contractions, first letters of string literals). There are two constraints on this: all the letters from the subtraction set must be in the target, and the length of the subtraction set must be less than the length of the target.

```
Subtraction → SubPreIndictator1 Wordplay (SubPreIndictator2) Wordplay
   | Wordplay SubMFUNCTION idIndictator Wordplay
   | Wordplay (SubPostIndictator1) Wordplay SubPostIndictator2
SubPreIndictator1 → “took”, “without” [...]
SubPreIndictator2 → “from” [...]
SubMidIndictator → “without” [...]
SubPostIndictator1 → “with” [...]
SubPostIndictator2 → “removed”, “deleted” [...]
```

Semantically here, we have the difference in pre- and post- as the difference between “wanted ant removed” and “removing ant wanted”

The letters in the set are thought to be removed in the order in which they’re found in order to be a properly clued wordplay. Thus “standing” with “tan” removed, gives “sanding”, whereas “ant” cannot be appropriately removed. Note though that the order in which nested clues are applied can change what the set is applied too. If we also had an anagram indicator, as in “Boy muddled standing missing trap” we can apply the muddled to standing to get “dansting” before removing “sting” to get the answer “Dan”.

3.6 Meta-references

Sometimes, clues contain references to that cannot be parsed in isolation, or contain a cluing structure that is incompatible with the main model of cluing.

3.6.1 Self reference

A type of clue called an ’&lit’ clue allows the setter to not include a definition part if the text that makes up the wordplay also can also be read as the definition. Thus in

```
Spoil vote! (4)
```

we have the wordplay Anagram (=spoil) “vote” to give ’VETO’, as well as the clue as a whole ’spoil vote’ meaning ’veto’.

3.6.2 Reference to other clues

Some publications will have clues that reference the answer to other clues (’8 across. Cake made badly by 7 down.’). Sometimes these may also be cyclical (in this example, 7 down would reference 8 across too).

3.6.3 Contextual References

Sometimes references will refer outside of the crossword itself. For example, The *Sunday Telegraph* on Easter Sunday 2014 had an anagram clue whose answer was EASTER SUNDAY, and its definition part was "today". In a crossword by setter *Araucaria*, "Araucaria is" coded for IAM (= "I am") as part of an answer.

4 Convention

Words

Part III
Naive Approach

5 Parsing and Evaluating everything

5.1 Solving a Clue

Our motivation here is to take a cryptic crossword clue, for example:

```
[A]  Ship carrying right flag (8)
[B]  Companion shredded corset (6)
and attempt to parse and solve it to provide the correct
answer.  We will define the datatype of Clue thus:
data Clue = Clue String AnswerLength
where
type Length = Int
```

In order to solve this clue, we want to find a function that takes a clue, which consists of a string containing the text of the clue and an integer representing the length of the required answer, and returns us the answer.

```
solve :: Clue → Answer
```

The intuition behind how our naive solver will work is that it will generated all possible ways of parsing a clue, then generate all possible answers that could be derived from those parses, and then attempt to match those up with the definition and the length constraints. In order to evaluate, measure and optimize each of these steps independently, we split the structure of our program into four parts:

```
solve = choose . evaluate . parse . split
```

where the types are given below:

```
split    :: Clue → [Split]
parse    :: [Split] → [Parse]
evaluate :: [Parse] → [Answer]
choose   :: [Answer] → Answer
```

5.2 Splitting

While a clue has a surface reading involving the semantic natural language parsing of it as a sentence fragment (which would yield a phrase, with an subject, a past tense verb and an object), we are only interested in the crossword interpretation of this, which is of the form:

```
  Definition Indicator* Wordplay
| Wordplay Indicator* Definition
```

Let us forget about the optional indicators for now – we will deal with these properly later . We are looking to define a function `split` which splits the clue into a wordplay portion and a definition portion. So for example, clue `A` can be split 6 different ways:

$$\overbrace{Ship}^{\texttt{wordplay}}\ \overbrace{carrying\ right\ flag}^{\texttt{definition}} \qquad \overbrace{Ship}^{\texttt{definition}}\ \overbrace{carrying\ right\ flag}^{\texttt{wordplay}}$$

$$\overbrace{Ship\ carrying}^{\texttt{wordplay}}\ \overbrace{right\ flag}^{\texttt{definition}} \qquad \overbrace{Ship\ carrying}^{\texttt{definition}}\ \overbrace{right\ flag}^{\texttt{wordplay}}$$

$$\overbrace{Ship\ carrying\ right}^{\texttt{wordplay}}\ \overbrace{flag}^{\texttt{definition}} \qquad \overbrace{Ship\ carrying\ right}^{\texttt{definition}}\ \overbrace{flag}^{\texttt{wordplay}}$$

From the types of `Wordplay` and `Definition`:

```
type Definition = String
type Wordplay = String
```

we can create a datatype

```
data Split = Def Definition Wordplay AnswerLength
```

as well as the signature of a function `split`:

```
split :: Clue → [Split]
split (text length) =
    let parts = partitions . words $ text
    in [Def (unwords d) (unwords w) length | [d,w] <- parts]
```

5.3 Parsing

Now we have consumed one portion of the string to form the definition in each of a list of splits. Now we need to parse the rest of the clue into a structure which we can evaluate to produce our answer. Let us take for an example the correct split (of the 6 available) of

$$\overbrace{Ship\ carrying\ right}^{\texttt{wordplay}}\ \overbrace{flag}^{\texttt{definition}}$$

which would have the Haskell structure of

```
Def ‘‘flag’’ ‘‘ship carrying right’’ 8
```

The `parse` function must, for each split, consume the wordplay and return all possible parses for that wordplay. Since each split will return multiple parses, we will want to collect these afterwards. We define datatype `Parse`:

```
data Parse = Parse Definition ParseTree AnswerLength
```

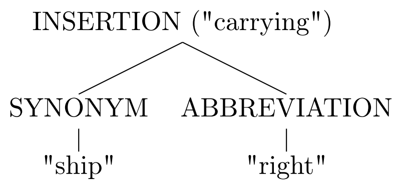
where `ParseTree` will be an Abstract Syntax Tree based on the structure of our clue. So we can now define:

```
parse :: [Split] → [Parse]
parse = concatMap parse’
```

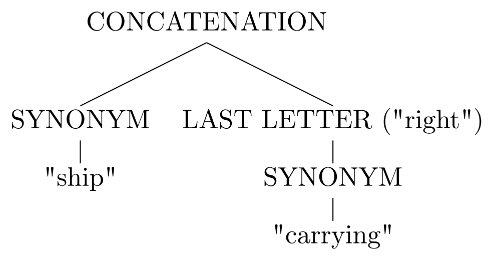
where

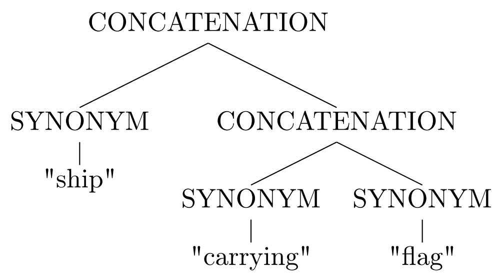
```
parse’ :: Split → [Parse]
```

In our example, we would require `parse’` to consume the string `Ship carrying right flag` to generate the parse trees – including the correct one:



as well as many others:





and so on.

5.3.1 Traditional Scanner-Based Parsing

The first step in the lexical analysis phase of a parse usually consists of tokenisation. This is the process of grouping characters together into functional groups called TOKENS, to later pass to the parser to perform the semantic analysis on. Tokens consist of a LEXEME the string of characters known to be of a certain type, and the value they represent (for example INTEGER 3 or VARIABLE NAME available_credit). The process is often split into two stages.

The Scanner The first is the SCANNER: this is often a finite state machine, which will consumer characters based on rules to produce potential lexemes. Some more simple scanners can operate under greedy assumptions (called the Maximal Munch principle by R.G.G. Cattel), and some require backtracking (for example, the language C). Due to the complexity and ambiguity of the language of cryptic crossword clues, it is not possible to produce an accurate scanner that produces anything other than a trivial tagging of lexical elements[2]

The Evaluator This stage of the tokeniser...

Scannerless Parsing Some parsers, traditionally often ones for simple languages such as...

Todo: need to finish off this section and explain why my parser is more like a scannerless parser. Also – this section is a bit incongruous here and kinda breaks the flow. Maybe it should be pushed elsewhere?

5.3.2 Parsing different clue types

So we need to define a function `parseClue` which will produce a parse tree from an unconsumed string. So we have

```
parseClue :: String → ParseTree
```

We will define parseString in terms of its parsing of various clue types, starting with one of the more simple unary ones:

```
parseAnagram :: String → [ParseTree]
parseAnagram xs =
 [AnagramNode (AIndicator x) y |
              (x,y) <- includeReversals . twoPartitions $ xs
                                        , isAnagramIndicator(x)]
```

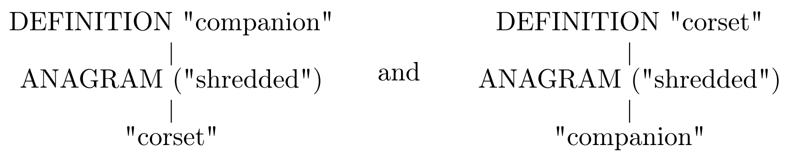
where

[2]It would, of course, be possible to produce a trivial parser for most languages, in which we lex every character or group of letters to a function with the value of itself, so instead of the desired VARIABLE X EQUALS INTEGER 3 we could instead simply parse to FUNCTION X FUNCTION = FUNCTION 3, and leave it to the rest of the pipeline to determine that FUNCTION 3 is a constant function which always yields the integer literal 3, but this misses the point of having a scanner separately.

```
twoPartitions xs = [(x,y) | [x,y] <- partitions xs]
includeReversals xs = xs ++ [(snd(x),fst(x)) | x <- xs]
```

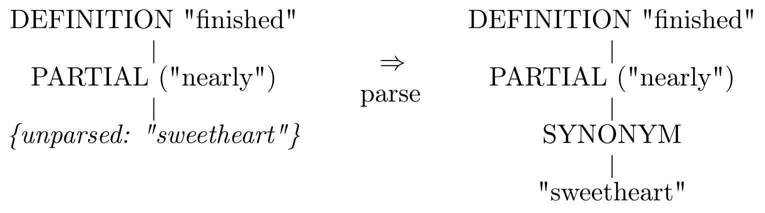
We allow both (x, y) and (y, x) through `includeReversals` in order to allow `muddled word` and `word muddled` both to indicate anagrams of “word”. This means a that in example [B] we parse both

.



.

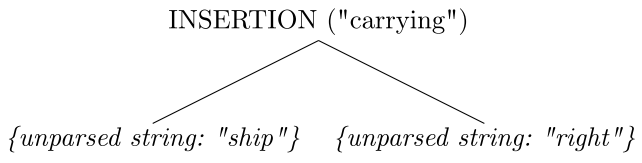
Anagram clues, along with Hidden Word clues only require a definition and a string, so their operands don’t require any further parsing. Other clues, though, may require the operands to be parsed. For example, the parsing of the clue `SWEETHEART NEARLY FINISHED (5)` as `(L)OVER` requires `SWEETHEART` to be parsed into a synonym node after we consumer nearly to be an indicator for a partial word node.



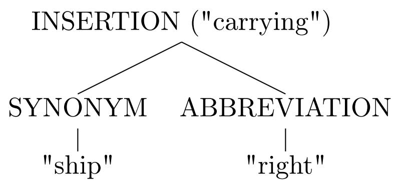
we therefore parse recursively, letting Haskell’s list comprehension take care of matching the correct partition to the correct parse.

```
parsePartialNode :: String → [ParseTree]
parsePartialNode xs = [PartialNode (LLIndicator x) y’
         |(x,y) <- includeReversals . twoPartitions $ xs
          , isPartialIndicator(x)
          , y’ <- parseClue y]
```

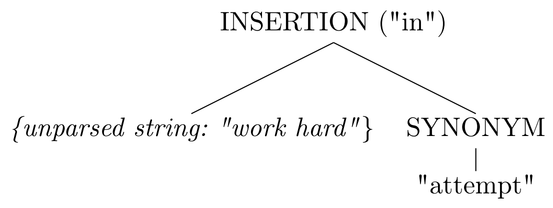
Still more complex clue types require splitting into three parts – two branches and an indicator – and often both of these branches require further parsing. For example, in the case of `SHIP CARRYING RIGHT FLAG`, choosing `FLAG` as the definition, we can generate



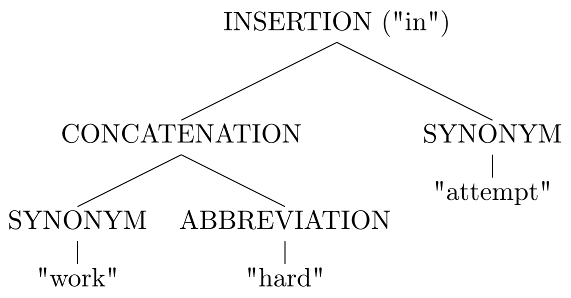
and then consume each of the unparsed strings in turn to produce



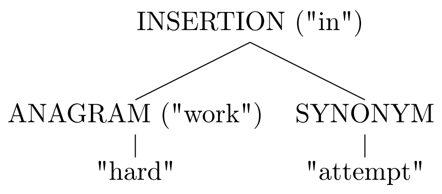
It is worth noting here that as well as the top-level parse generating multiple different options, each of these sub-parses may also generate several different parses, and these themselves may be complex with multiple sub-parses. In the clue `WORK HARD IN ATTEMPT TO GET CUP`, with definition (=’’`to get`”) of `CUP`, we can parse the wordplay as



which may subsequently evaluate to the (correct, in this case) parse:



as well as others, such as:



Here, again, we allow Haskell’s list comprehension take care of constructing the sub- parse-trees from our recursive calls and constructing them into our final list of trees.

```
parseInsertionNodes :: [String] -> Int -> [ClueTree]
parseInsertionNodes xs n = let parts = threeParts xs
      in [InsertionNode (IIndicator y) x’ z’
          | (x,y,z) <- parts, isInsertionWord(y)
            , x’ <- (parseClue x n)
            , z’ <- (parseClue z n)]
```

5.3.3 Parse Multitudes

These nested parses can cause the number of produced parses to grow exponentially as the depth of the nesting increases. Longer strings with more indicators – especially indicators for binary expressions such as insertion indicators and subtraction indicators – are more likely to produce deeply nested parses, and therefore return a large number of parse trees.

5.4 Evaluation

In the evaluation stage we look to define a function `evaluate`with the type signature:

```
evaluate :: [Parse] → [Answer]
```

As the evaluation of each `Parse` will yield a list of multiple `Answer` (e.g. an anagram node of a five-letter word will evaluate to 120 different answers, although very few of them will be valid words), we can define `evaluate`as

```
evaluate = concatMap eval
```

where `evaluate` will consume `Parse` data in the form `Def Definition ParseTree AnswerLength`and produce `[Answer]`, where:

```
data Answer = Answer String Parse
```

The parse is included along with the answer, as it contains the definition for that parse, which will later allow us to check that our generated answer has some relation to what we thought we were looking for in that parse, and also allows us to reconstruct the reasoning behind the clue by inspecting the parse tree.

We then define `eval` as:

```
eval (Def d pt l) = [Answer x (Def d pt l) |
                                          x <- evalTree pt]
```

We can then define `evalTree` in terms of the different types of node in our `ParseTree` type. Those without subtrees will be defined simply with reference to a Haskell function that performs their action:

```
eval_tree (AnagramNode ind xs) c = anagrams xs
eval_tree (SynonymNode xs) = synonyms xs

anagrams :: String -> [String]
anagrams [] = [[]]
anagrams xs = [x:ys | x<- nub xs, ys <- anagrams $ delete x xs]

synonyms :: String -> [String]
synonyms xs = Map.lookup xs thesaurus
```

and so on. Clues with sub-trees are treated with a similar recursive call, with either a map, or a list comprehension applying the expressions function to each generated sub-answer

```
eval_tree (ReversalNode ind ys) = map reverse (eval_tree ys)
eval_tree (ConcatNode ind xs ys) = [x ++ y | x <- eval_tree xs
                                           , y <- eval_tree ys]
```

Todo: there's probably space here to give each definition, since this really is the bread and butter of the whole affair

5.5 Selection

Finally, given that we've produced our list of answers, most of which will be meaningless combinations of jumbled letters and synonyms pressed together, we need to filter down to the answer containing a string which in some way meets the criteria set for us in the clue, that is 1. finding an answer that is a synonym of the part of the clue we chose as the definition 2. being the right number of letters.

So we can define

```
choose :: [Answer] → Answer
choose = head . filter valid

valid (Answer ans (Def def pt len)) = (length ans == len)
                                      && (is_synonym ans def)
```

Of course, we may not have generated a valid solution, so we can redefine to include this uncertainty:

```
choose :: [Answer] → Maybe Answer
choose = headM . filter valid
```

6 State space and performance analysis

6.1 Overview - does it work?

This approach has the required structure to correctly parse and solve most cryptic crossword clues — with some caveats.

Firstly, although in most cases the correct parse was generated, often the number of other parses to be evaluated before reaching the correct one was so great that the computation would effectively not end. In this case, the heap size wasn't continually growing, as each evaluation branched and then diminished in turn, but the running time was sufficiently large (>48hrs) such that the computation would be useless in a practical situation. The data for this is considered in **6.2**

In other cases the correct parse was created, however the semantic data wasn't available to evaluate the clue correctly. In other, very rare cases, there is a clue which does not fit the structure of the grammar defined in **Part I**. These do not generate the correct parse trees, and so are not soluble. These are discussed in **6.3**.

6.2 Correctly parsed and evaluated clues

Most clues, if they yield any results at all, yield them within 30 seconds of being run. Many others yield them a very long time afterwards – multiple hours of runtime is required to reach them. Others seem to run indefinitely.

Of those that do not terminate within an acceptable timeframe, the generated parse trees can be inspected and it can be shown that the correct one has been generated, and that since no individual evaluation takes infinite time, and each evaluation uses a non-problematic amount of stack space (that is to say – the stack does not increase with each subsequent evaluation), then we can say that the clue is solvable, even if not in a reasonable amount of time.

The clue `FRIEND FOUND IN OKLAHOMA TERMINAL (4)` yields the correct parse:

```
Def "friend" (HiddenWordNode (HWIndicator ["found","in"])
                                         ["oklahoma","terminal"])
```

however it also generates 59 others, including:

```
Def "friend" (InsertionNode (IIndicator ["in"]) (Leaf "found")
                (ConsNode (Leaf "oklahoma") (Leaf "terminal")))
Def "terminal" (InsertionNode (IIndicator ["in"])
            (Leaf "friend found") (Leaf "oklahoma"))
Def "oklahoma terminal" (ConsNode (Leaf "friend")
                                  (Leaf "found"))
Def "terminal" (ConsNode (Leaf "friend") (ConsNode (Leaf "found")
                (ConsNode (Leaf "in") (Leaf "oklahoma"))))
Def "terminal" (ConsNode (ConsNode (Leaf "friend")
                         (Leaf "found")) (Leaf "in oklahoma"))
Def "friend found" (Leaf "in oklahoma terminal")
Def "in oklahoma terminal" (ConsNode (Leaf "friend")
                                     (Leaf "found"))
Def "in oklahoma terminal" (Leaf "friend found")
```

While evaluation of the correct parse takes 0.05 seconds, the evaluation of the first of the other examples takes over 10 seconds - it is the cumulative effect of the evaluation of the others, as well as the order in which they appear in the list which determines how long the total solving time takes. Some of these effects can been seen in **Figure 1.**

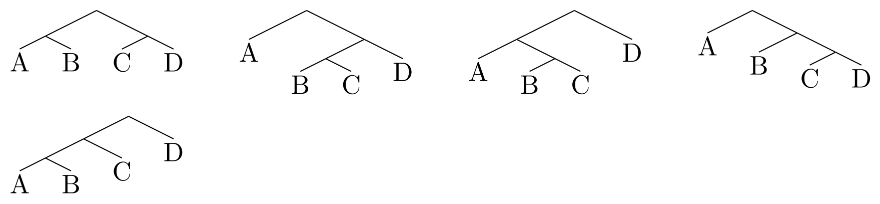
Clue	**Solution**	**Clue Length**	**# Parses**	**# Evaluations**	**Solve Time**
COMPANION SHREDDED CORSET (6)	ESCORT	3	8	148,500	0.2s
HOPE FOR HIGH PRAISE (6)	ASPIRE	4	25	105,718,343	1.39s
MARIA NOT A FICKLE LOVER (9)	INAMORATA	5	60	84,855,252[2]	—[1]
FRIEND FOUND IN OKLAHOMA TERMINAL (4)	MATE	5	59	92,995,844[2]	—[1]
PAUSE AT THESE I FANCY (8)	HESITATE	5	54	5,358,615	4.59s
ANKLE WAS TWISTED IN BALLET (8)	SWAN LAKE	5	84	203,991,525	12.13s
NOTICE SUPERVISOR IS GOING NUTS AT FIRST (4)	SIGN	7	853	—[3]	—[1]
ATTEMPT TO SECURE ONE POUND FOR A HAT (6)	TRILBY	8	2930	—[3]	—[1]

Figure 1: Solving statistics for selected clues on a 2014 MacBook Pro

[1] Although the correct parse was generated, and selective evaluation of that parse yielded the correct results (i.e. a solution would be available eventually), the normal solving procedure did not compute the correct answer within 48hrs of running time
[2] Due to Haskell's lazy evaluation, this can sometimes be calculated without actually computing the solution
[3] Could not yield answer within 48hrs

6.2.1 The effect of clue length on the number of parses

The length of the clue has an exponential effect on the number of parses produced. This is due partly to the increasing number of ways in which binary trees can be constructed from N elements, as in:



It also increases the availability for function words to interact with each other - when any A, B, C, or D in the examples above also have multiple parses, this is when we see the strongly trended exponential growth seen in **Figure 2** (displayed on a logarithmic scale).

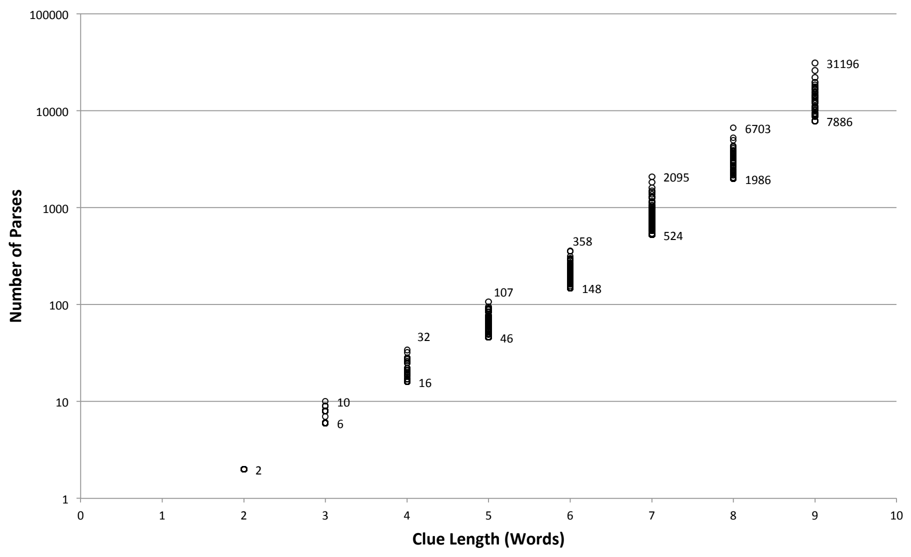


Figure 2: Number of parses generated for varying clue lengths over 600 sample clues

6.2.2 The effect of clue length on the number of evaluations

We see a similar but even greater effect on the number of evaluations, with the effect of the exponential growth per parse compounded by the fact that each parse can evaluate out to to thousands of options. This is due to two effects, Firstly, clues types like anagrams can have many thousand evaluations per parse (there are 120 anagrams of a 5-letter word, rising to 40,320 anagrams of an eight letter word). Secondly, compound clues like insertions, which can take the result of one wordplay and insert into the second, can magnify the effect of branching in its sub-clues.

There are 4 ways that a word 'A' can be inserted into a 5-letter word 'B'. There are 480 ways that it can be inserted into each of the 120 anagrams of word 'B', and if there are also 120 different anagrams of word 'A', then there are 57600 different evaluations for that parsed arrangement.

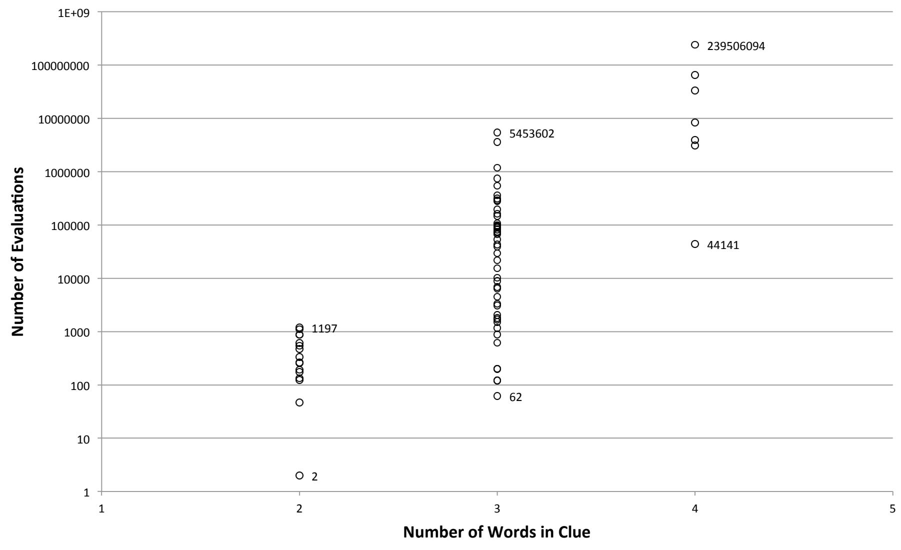


Figure 3: Number of evaluations generated for varying clue lengths over 75 sample clues

The size of the thesaurus has a large impact on the number of evaluations produced, as all clue types (other than Anagram, Hidden Word and Initials, which use String) use Synonym as the lowest level node in their sub-trees.

Figure 4 (also displayed on a logarithmic scale) shows how limiting the number of synonyms returned by the thesaurus affects the number of evaluations. The graph plateaus as the restriction exceeds the actual number of synonyms per entry for each word in the thesaurus.

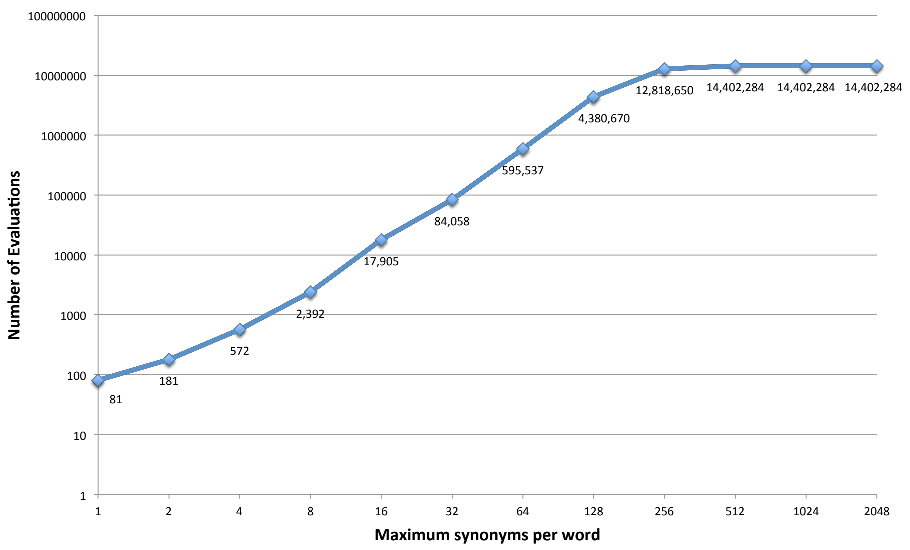


Figure 4: Number of evaluations generated by restricting the maximum thesaurus length for the clue "Good opportunity in school" (5)

6.3 Analysis of selected clues which are not correctly solved

It is difficult to perform a large-scale analysis of the numbers of clues for which the data does not exist, or where the correct parse is not generated, as often these will present themselves in the same way as the correct clues with too large a search space, that is by not terminating within an acceptable time.

These clues are therefore presented as an illustrative sample of the sorts of errors that prevent correct parse (**6.3.4**) or correct evaluation (**6.3.1 – 6.3.3)** being generated.

6.3.1 SHINY SILVER PAPER IN THE STREET (8) (= "AGLITTER") [Guardian]

Although the correct parse is generated ([SILVER] + [PAPER IN THE STREET]), some natural language analysis would be required to derive the fact that "PAPER IN THE STREET" = "litter"

6.3.2 PLAYWRIGHT AT HOME HAVING CAUGHT DISEASE (5) (="IBSEN") [Everyman]

This clue requires two pieces of category knowledge, firstly that Ibsen is a member of the set of playwrights (and not a synonym for playwright), and that BSE is a member of the set of diseases

6.3.3 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.

6.3.4 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:

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

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

7 Algebraic + computational simplifications

7.1 Pruning out equivalent trees (Canonicalization)

7.2 Discussion of lazy evaluation

7.2.1 e.g. Anagram Nodes - should we pre-compute and thread through the program?

7.3 Analysis of improved solution with respect to state space etc.

8 Further heuristic simplification -

8.1 matching the Human Solver's thought process

8.2 Constrain length while parsing maximum and minimum

8.3 Constrain length while solving

8.4 Constrain against known letters

8.5 Constrain against known wordlist

8.6 Constrain against synonyms

8.7 Analysis of improved solution with respect to state space etc.

9 Solving based on most probable parse trees

9.1 Evaluation functions

9.1.1 Current weighting just 'works'

9.1.2 Could generate weighting based on ML + cost minimization

9.1.3 Providing a weighted list of possible answers

10 Analysis of Single clue solving against benchmarks

10.1 Accuracy given data Accuracy

10.2 assuming reasonable data could be acquired

10.3 Computation required and feasibility in the real world

10.4 Discussion of parallelization

Part V
Future Work

The Generation of Cryptic Crossword Clues G. W. Smith, and J. B. H. du Boulay - 1986

Crossword Compiler-Compilation H. Berghel and C. Yi. - 1989

PROVERB: The Probabilistic Cruciverbalist Greg A. Keim, Noam M. Shazeer, Michael L. Littman - 1999

Computer Assisted Analysis of Cryptic Crosswords P.W.Williams and D. Woodhead - 1977

LACROSS language, formal definitions - good building material Cryptic crossword clue interpreter M Hart, RH Davis - 1992

Microcomputer compilation and solution of crosswords RH Davis and E J Juvshol - 1985

Give Us A Clue Jon G. Hall and Lucia Rapanotti - 2010

A Statistical Study of Failures In Solving Crossword Puzzles Naranana, 2010

Expertise in cryptic crossword performance Kathryn Friedlander, Philip Fine, 2009

Cattell, R. G. G. “Formalization and Automatic Derivation of Code Generators”. PhD thesis, 1978. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA