Birkbeck College, University of London

School of Computer Science and Information Systems

MSc Computer Science

Project Report

Solving Logic Problems through the Use of

Specialised Language Parsers

Supervisor: Michael Zakharyaschev

Author: Abigail James

This project is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

2013

# Abstract

This project has been an investigation into the use of specialised parsers to improve the success rate of specialist tasks by handling semantic ambiguities that general parsers would stumble over. Logic problems with natural language clues have been chosen as the studied specialist task as current solutions have automated their solution from pre-translated clues, but not with perfect success when handling the prose clues themselves. Once translation to functional rules is achieved, the investigation turns to re-translating the clues to statements of first order logic.

# Contents

Background to the Specialist Task

Program Requirements

Schedule

Design Considerations

Implementation

Evaluation

Summary

# Background to the Specialist Task

## Task Suitability

The topic of this project was chosen whilst considering the current ability of software to solve a form of logic problem the author is very familiar with, commonly known as Grid Logic Problems. Existing software can be found that will solve these problems, or ones of a similar format, however some require a human user to translate the natural language clues, and one other could translate the clues, but through use of a general parser found situations in which the full meaning of the clue could not sufficiently be translated as to allow the logical unit to solve the problem. Thus it was decided to investigate the possibility of developing a specialist parser that could more readily handle the particular subject problems.

## Logic Problems

The logic problems considered evolved from an older format, popular in the fifties after publication of the most famous example, known as the Einstein Puzzle, included here.

1. There are five houses.

2. The Englishman lives in the red house.

3. The Spaniard owns the dog.

4. Coffee is drunk in the green house.

5. The Ukrainian drinks tea.

6. The green house is immediately to the right of the ivory house.

7. The Old Gold smoker owns snails.

8. Kools are smoked in the yellow house.

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house.

11. The man who smokes Chesterfields lives in the house next to the man with the fox.

12. Kools are smoked in the house next to the house where the horse is kept.

13. The Lucky Strike smoker drinks orange juice.

14. The Japanese smokes Parliaments.

15. The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

At the core of the puzzle is a collection of categories, each containing a certain number of named items. Typically there is a named individual, or in the above case a person of a particular nationality. Each individual is related to just one item in each other category, as partially revealed by the clues. For example, the Ukrainian is known to drink tea. The idea of the puzzle is, by process of elimination and perhaps some reduction to absurdity, to identify who relates to the items in the final question, which won’t necessarily have been explicitly mentioned, but should be identifiable by considering all clues.

The modern, popular format that this project uses for the ready supply of different puzzles, is often referred to as a “grid logic problem” due to the solving aid that is provided alongside the clues to assist the user in identifying all the relations within the puzzle. Because of this grid, all of the category items are specified beforehand, so there is no need to be able to semantically group Norwegian, Frenchman, etc. into the category “Nationalities”. The linguistic style of the clues is also somewhat standardised, so creating a specialist parser is not as complex a task as creating a general-purpose language parser, and therefore a good mix of parsing/solving could take place within the confines of this project.

Once the natural language clues have been parsed, the relations can be used to match every item from the first category to an item in every other category, so this type of logic problem could be considered more complete than an Einstein puzzle as it requires a full investigation not limited to just one or two of the provided categories.

## Human Solving Methods

The human user will consider each clue of the puzzle in turn, normally, and mark the grid with any item matches or mismatches that the clue reveals. An example problem is now considered.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Brown | Green | White | 7 | 9 | 10 |
| Anne |  |  |  |  |  |  |
| Brian |  |  |  |  |  |  |
| Mary |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |

### Example Probleminsert footnote reference

Three children live in the same street. From the two clues given below, can you discover each child’s full name and age?

1. Miss Brown is three years older than Mary.
2. The child whose surname is White is 9   
   years old.

Solution:

Miss Brown (clue 1) cannot be Brian, so a cross is entered in the Brian/Brown box. She cannot be Mary, so a cross is placed in the Mary/Brown box. This means she is therefore Anne, so a tick is placed in the Brown/Anne box and further cross marks placed in the other potential surname boxes in Anne’s row.

Anne Brown is three years older than Mary (clue 1). She must be 10 and Mary 7. Ticks are placed to represent this in the forename/age area of the grid and corresponding crosses against mismatching values. In the surname/age of the grid, a tick and corresponding crosses can mark the Brown/10 pair.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Brown | Green | White | 7 | 9 | 10 |
| Anne | *y* | *x* | *x* | *x* | *x* | *y* |
| Brian | *x* | 🗶 | ✓ | *x* | ✓ | *x* |
| Mary | *x* |  | 🗶 | *y* | *x* | *x* |
| 7 | *x* |  | 🗶 |  |  |  |
| 9 | *x* | 🗶 | ✓ |  |  |  |
| 10 | *y* | *x* | *x* |  |  |  |

The grid then reveals that Brian must be 9, so a tick can be entered.

Clue 2 states that White is 9, so that must be Brian. Ticks are entered in the White/9 and White/Brian boxes and for each, corresponding crosses.

This leaves Green as the only remaining possible surname for Mary. Filling in a tick here completes the grid.

### Using the Grid

For a problem of the size above, the grid might not seem necessary once the formation of the puzzle is understood. With just three items per category and only three categories, it can potentially be solved mentally with a small amount of concentration, however these problems can expand both in terms of size and complexity. The most readily found problem consists of four categories with five items in each, with larger ones provided occasionally for those looking for a problem that will require a longer length of time to solve. The increased size means that more hints are needed, but also that more time must be spent considering other relations found by investigating the grid, rather than the original clue. For example in this partially completed grid we have deduced that A1 cannot be D4 or D5, noted by the crosses in the boxes for A1/D4 and A1/D5. It can also be seen that A1 has been shown to be connected to C4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | B1 | B2 | B3 | B4 | B5 | C1 | C2 | C3 | C4 | C5 | D1 | D2 | D3 | D4 | D5 |
| A1 |  |  | 🗶 |  |  | 🗶 | 🗶 | 🗶 | ✓ | 🗶 |  |  |  | 🗶 | 🗶 |
| A2 | 🗶 | 🗶 | ✓ | 🗶 | 🗶 |  |  |  | 🗶 | 🗶 |  |  |  | 🗶 |  |
| A3 |  |  | 🗶 |  |  |  |  |  | 🗶 |  | 🗶 | 🗶 | 🗶 | ✓ | 🗶 |
| A4 |  |  | 🗶 |  |  |  |  |  | 🗶 |  |  |  |  | 🗶 |  |
| A5 |  |  | 🗶 |  |  |  |  | 🗶 | 🗶 |  |  |  |  | 🗶 |  |
| D1 | 🗶 |  |  |  |  | ✓ | 🗶 | 🗶 | 🗶 | 🗶 |  |  |  |  |  |
| D2 | 🗶 |  |  |  |  | 🗶 | 🗶 | 🗶 | 🗶 | ✓ |  |  |  |  |  |
| D3 | 🗶 |  |  |  |  | 🗶 |  |  |  | 🗶 |  |  |  |  |  |
| D4 | ✓ | 🗶 | 🗶 | 🗶 | 🗶 | 🗶 |  |  |  | 🗶 |  |  |  |  |  |
| D5 | 🗶 |  |  | 🗶 |  | 🗶 | 🗶 |  |  | 🗶 |  |  |  |  |  |
| C1 |  | 🗶 |  | 🗶 | 🗶 |  |  |  |  |  |  |  |  |  |  |
| C2 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C3 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C4 | 🗶 | ✓ | 🗶 | 🗶 | 🗶 |  |  |  |  |  |  |  |  |  |  |
| C5 |  | 🗶 |  |  |  |  |  |  |  |  |  |  |  |  |  |

By understanding that all items related to A1 are now related to C4, we can complete C4/D4 and C4/D5 boxes, and in fact deduce that A1 and C4 must be connected to D3. This in turn leaves the only possible C connection for D5 as C3. By continuing this process the whole of the above grid can in fact be completed.

## Linguistic Element

The linguistic complexity of the clues provided can vary. For example, clue 2 in the example problem illustrates a simple, direct relation, whereas clue 1 provides a comparative relation between two items. Full clues will normally include a collection of these two, or could perhaps provide a subset of items within which one individual’s item can be found (“Samantha’s surgeon was a woman”). The latter example clue hints at one of the extension tasks of this project – to provide basic semantic interpretation of a few more commonly-occurring scenarios. Puzzles do occasionally rely on the reader to be able to identify a name as either male or female and upon discovering the mention of this subset identifier, it is possible for the software to be able to request the user gender-categorizes available names.

# Program Requirements

The central aim of the project were to use a test-driven development approach to produce a modularised application capable of solving logic problems from natural language clues. The specialised parser would form one of the modules, and the logical deduction of the solution would take place in a separate module. Beyond this, there were some itemised requirements that the resultant software should meet, namely that it:

* should load problems straight from input source without assistance
* should translate the natural language clues as-provided
* should use the translation of the clues to solve the problems correctly
* should present both the original puzzle text and the solution found to the user
* should report the time taken to solve each puzzle.

Additionally, if time allowed, a couple of extension items were planned:

* The software should retranslate the clues to first-order logic statements and pass these along with the solution to a first-order theorem prover to verify correct deduction of the problem’s solution.
* For semantic limitations, the software should prompt the user for assistance in categorising some items, for example, stating which names out of a collection were female and which male.

# Project Development Plan

The project was broken into three main stages: building the module capable of solving problems from pre-translated clues, building the module capable of translating the clues, and finally creating a user interface and investigating the use of a first-order theorem prover to verify the solution uncovered by the logical module. A phased software development methodology was therefore chosen as the most appropriate approach to take in developing the program. The test-driven approach would also be very important in verifying the success of the logical and parsing modules, and so would control the development undertaken within each stage of work. The planned schedule follows.

|  |  |
| --- | --- |
| Dates | Targets |
| Apr 9 - May 9 | Build up the bank of logic problems and their solutions. Create method to import problems. |
| May 9 – Jun 3 | Build initial dictionaries of conjunctions and comparative terms by studying puzzle clues. |
| Jun 3 – Jun 17 | Create training statements and first methods in solving algorithm (positive/negative associations). |
| Jun 17 – Jun 24 | Create methods for process of elimination and absurdity reduction checkers. Create user-friendly output of solution progress. |
| Jun 24 – Jul 1 | Create “problem solved” detection method and containing class to solve problem from training statements. Include timing of solving methods. |
| Jul 1 – Jul 15 | Create method of transforming real puzzle hints into bare associations through use of dictionaries. |
| Jul 15 – Aug 5 | Improve sophistication of text interpretation module. Add GUI elements to show solution progression. |
| Aug 5 – Aug 19 | Transform clues to FOL statements for FO prover. |

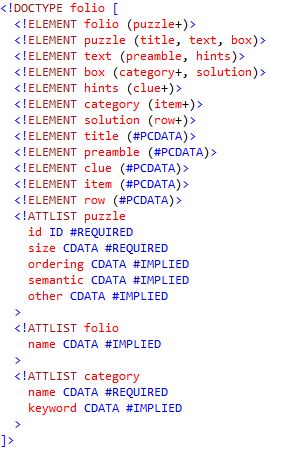
# Design Considerations

## Development Environment

The software would be developed in Visual Studio 2012, with code in C#. This was chosen due to familiarity with the IDE and language, whilst also retaining the option to write modules (the Logix module in particular) in C++, if so desired. The chosen third-party first-order prover was also developed in C++ and therefore should be includable as a project within the whole solution from the source code. The user interface would be a simple collection of WinForms, as the presentation was not the main focus of the project.

## Puzzle Input Format

For the input format of the logic problems, XML representation was chosen to allow a whole collection of problems to easily be imported by the software. The puzzle source was defined by the below DTD. The XML document containing the puzzles, once transcribed, was tested against the DTD using an online XML validator. This helped to identify a few transcription errors that may have caused difficulties.



The names for individual elements of the XML document were chosen based upon the visual representation of the problems in the original sources, which were two editions of the Logic Problems magazine. The *ordering*, *semantic* and *other* attributes were added during transcription as a way of gauging the complexity of the problem. They are not used by the program, just the author when deciding which puzzle to use for each test.

## Program Structure

The logical and parsing capabilities of the program were modularised into individual units so that one could be used without the other in any extension activities. A main, controlling module would call each unit as required to solve a puzzle. The particular forms used to represent the puzzle and relational information was contained in another module, which all three other units refer to. Thus the dependency diagram can be expressed rather simply:

CleverZebra

Logix

Parser

Representation

## Relation Representation

A rather simplistic method of representing relations was used, as featured in the project proposal. If the first item in the first category was identified as being related to the first item in the second category, for example, the resulting relation would be “A1=B1”. If they were identified as not being related “A1!=B1” would represent this. A comparative relation, i.e. one category item’s corresponding item in another category is expressed as being more/less than another item’s, would be represented as “A1(B)>C1(B)”. If the exact difference between the two items is known, this would be shown in the form “A1(B)-C1(B)=2”. Later, when considering the possibility of semantic categorisation, the rule format “A1(B)={male}” was added. A representation for conditional statements was also chosen, even though the chosen puzzle source does not feature these types of clues, for the sake of completeness. An example could be “?A1=B1?A1(C)>B3(C):A1(C)>B1(C)”, arising from “Mr Jones’ bought a tree the day after the other tree was bought” where there are two items in category B that are types of trees.

# Implementation

## Stage One – Logix

After creating the first project within the Visual Studio solution, CleverZebra, the project for the logical module was created and the name Logix chosen. An additional project was created to house the tests, LogixTests. The first test was written in C# and then development began within Logix to create classes to allow the test to pass. The first burst was written in C#, though timings were noted at this early stage. If they seemed disappointingly slow, then development would have switched to C++, however results were passing with promisingly good times, so work continued in C#.

### Classes

The first development work focussed on the construction of the class that would hold all information relating to one category. It would have an array for positive matches, an array for negative matches and an array for the values of its items. As the whole class revolved around the need to store the information, the class was originally called Line, after a line of the grid solving aid, but as the code became more substantial and the grid less important, this was renamed to the **Category** class.

Category example {

size = 3,

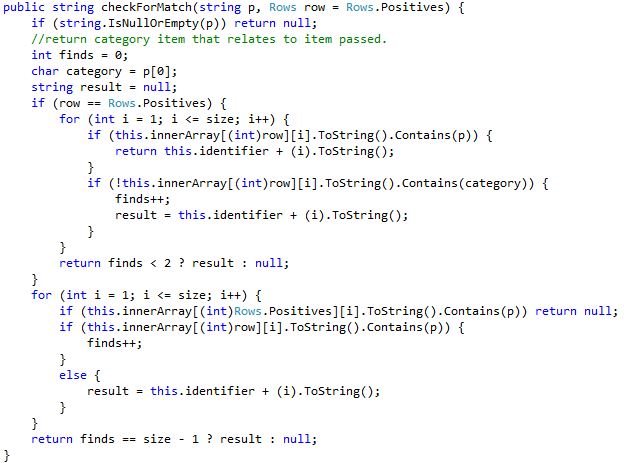
identifier = ‘C’,

keyword = “days”,

innerArray = { {0,1,2,3}, {‘A’, “Monday”, “Wednesday”, “Friday”}, {“+”,””,””,””},   
{“-”,”A1”,”D2”,”A3”} }

}

The purpose of the first three tests to be written was to check the creation of an object of this class and two of its basic methods. The Category class would need to receive a whole relation and be able to return whether it used this relation and also whether information held within the class object could provide any further relations, given the one considered. The name of the second test hints at this purpose – Test\_AllButOneFound. This simulates a Category ‘B’ object for a puzzle four items deep and uses the “addRelation” method to mark a list of pairs – A1=B1, A2=B2,A3=B3,A4=B4. It then calls “checkForMatch(“B5”)” and this returns the string “A5”. This method is considering, for a given item, whether enough information for that item’s category has been recorded to match the item to its associated item with the B category. (If the item has already been listed in the positive row of the innerArray, that index is returned with the category identifier to provide its already-known match.) The method also considers whether the item to be checked has been listed as a non-match against every item bar one, thus revealing the only possible item remaining. The result of the checkForMatch function is checked after it is called and a new relation formed if a matched item is found.



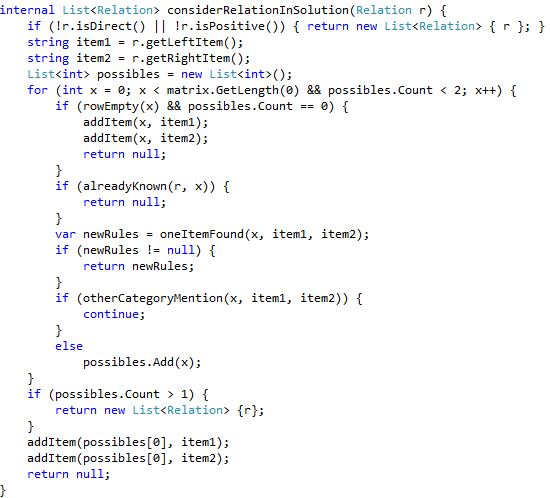
The relations themselves were originally held in one class, with the intention of analysing the held string when needing to interact with other classes depending on whether the relation was direct, comparative, etc. However this was soon separated into a collection of **Relation** classes inheriting from the base Relation class. By project end, the relation types created were DirectRelation, RelativeRelation (for comparative relations), ConditionalRelation and SemanticRelation. Each Relation type held a particular number of items and overridden functions such as getBaseItem() to return the correct item required by context, given its sort. A **RelationFactory** class was created to instantiate the correct type of relation based upon the given string. The RelationFactory can also build a DirectRelation from the individual parts, as this is often required by methods generating newly-discovered relations. For the purpose of maintaining the level of abstraction necessary to allow the modules to function independently, the RelationFactory references the external Representations project when deciding what sort of Relation to create.

Another method created in the early tests is the considerRelationToCategory method, which would be one of the most crucial functions within the Logix module. Similar to the checkForMatch, method, but operating on the level of a whole relation, rather than an individual item, considerRelationToCategory checks whether the information held within that relation should be stored in the subject category and whether it reveals any new relations in conjunction with information stored within the category. The method contains a pathway for each type of relation and returns either a collection of relations generated by considering the input relation, just the input relation itself, if nothing can be discovered within this category, or an empty collection if the category has used the input relation in the only place it should be used (i.e. for a direct relation featuring the subject category).

To handle comparative relations, a family of classes derived from a base **Calculator** class were added. These classes can put the relative difference between two items into context of the actual values available and instantly create negative relations to any items which this rule alone shows are impossible partners for the two items mentioned. Once an item actually has an assigned value within that category due to other rules, the calculator then determines which item the second must be related to if the relation is quantified (A1(B)-C2(B)=5) or to any values which are higher/lower than the second lower/higher item can be.

After these first few tests of the Category class, the class that actually holds the main solving algorithm was created: the **Deducer** class. A Deducer object is instantiated based on a given puzzle and contains an assortment of functions to control the process of finding the solution to that puzzle and also to fire events that can update the UI with progress.

A **Solution** class was created to record the positive matches found that would be mirrored in the UI and also to cover the situation where a pair of relations can only be related to other categories by considering existing relation pairs. When the Solution class receives a positive DirectRelation (the only sort of Relation it considers), it tries to fit this pair of related items into the table of matches if it can.



The matrix is the Solution’s internal representation of the answer grid with one row per item in each category. The method considers each row of the matrix in turn, looking to fit the two matched items into the grid, if possible. So if the relation “A1=C3” was to be considered, then depending on the state of the Solution, different resulting relations might be returned.

If the first row is empty (i.e. nothing has yet been recorded), the two related items in the relation begin considered are added to this row.

|  |  |  |  |
| --- | --- | --- | --- |
| **A1** |  | **C3** |  |
|  |  |  |  |
|  |  |  |  |

Otherwise if both items are already contained within that row, we exit the function, returning no new relations. Otherwise if one of the items is found in that row, then the second is added and positive relations between it and any other items in that row are created.

|  |  |  |  |
| --- | --- | --- | --- |
| *A1* | B4 | **C3** |  |
|  |  |  |  |
|  |  |  |  |

If neither item is found in that row, but a different item of the same category as either is populated then the rule cannot be placed in that row. If no other item of either category is mentioned, then that row is considered a possible fit for the input rule. Once all rows have been considered, if the process has not already returned new relations and it has found only one possible row for the input rule, it now places the items into that row and creates any new relations revealed (e.g. “A1=B4”).

|  |  |  |  |
| --- | --- | --- | --- |
| A2 | B3 |  |  |
| **A1** | B4 | **C3** | D1 |
| A3 |  | C2 |  |

The final class diagram for Logix is:

[create a diagram!]

### The Solving Algorithm

From the start, one of the aims of this stage of the project was to consider a couple of ideas for solving algorithms and determine which one had more success. The two separate ideas came from thinking about methods used by humans to solve the puzzles. You can start from the beginning of the clues, add in the basic relational information these contain and then when the clues are exhausted look for information highlighted by the grid to uncover new relations and effectively then mark those until the whole grid is filled in. The other tactic is to look for these additional relationships each time a new relation is added to the grid. This helps to keep more relevant information in mind to spot and include new relations whilst working through the clues. For the computerised algorithm, therefore, the idea was to do an initial pass of all clues and then consider others found, or to start looking around for these additional relations between each input clue. The relevant functions for the algorithm lie within the Deducer class. The simply named Go() function kicks everything off:



[walkthrough this function]

Originally the ABSURDIO\_SPACING was not present and a check for additional relations was made every turn. Once the algorithm had been used to successfully pass the first test, however, optimisation began, which included deciding at what point the Absurdio function should be used. Absurdio being called at every pass resulted in the first test problem, containing four categories each containing three items, being solved in 52 turns. As the whole problem at this size only contains 54 possible pairings ((3\*3\*3)+(2\*3\*3)+(1\*3\*3)), this seemed pretty poor. Following the original plan for the second algorithm, based on human solving methods, the ADSURDIO\_SPACING value was set to 5 – the average number of clues in these puzzles, meaning it would first be considered once all main clues had been placed. This immediately dropped the number of turns required to 37. Experimenting with other values resulted in a nonlinear relation: increasing the spacing improved results up to a point, but reached a maximum value at around 8, which was therefore chosen as the value to use.

Timing of the tests had shown that the problems could be solved in time intervals so small, that the real comparison was in the number of turns, and not timing. Originally every relation was compared to the individual categories and then separately to the solution that was being developed. When looking for ways to optimise the Deducer’s progress, however, an investigation into the use of events to simultaneously update the solution as a positive item is added to or created by a category turned out to be very fruitful. The third test puzzle was the first of size 4x4 to be considered and it was being solved in a disappointing 106 turns and ~80ms. Adding in an event trigger for the solution instantly reduced this to 55 turns and a solve time of 8ms.

(Jul 13th added addInverse rather than adding a negative relation to the clue stack. Much better idea.)

## Stage Two – Parser

Whereas for Logix the inspiration for code design came from knowledge of manual processing methods, the influence for Parser’s methods came from reading about the subject of natural language processingAdd references. The most important concepts learnt were the use of “tagging” to categorise each word in a sentence as a particular part of speech, dictionaries to allow the tagging to take place and a buffer to match to grammatical rules based on the sequence of individual tags. This project did not require the sentences to be matched to rules of natural language grammar, but rather to expected patterns in the more restricted grammar of puzzle clues. Not every word would be tagged, just those relevant to the problem type generally and then additionally those that are relevant to the particular problem. This meant there would be one largely static dictionary and another created per puzzle.

### Structure

Mirroring the structure of the whole program, the **Parser** class was developed to call the **Tagger** and **Translator** classes in turn. The Parser is instantiated with the puzzle, which it uses parts of to then in turn instantiate the Tagger and Translator. The Tagger creates a **TermsDictionary** which contains words required by every puzzle and which considers keywords within the puzzle to include additional words that might also be necessary. For example, if a category carries the keyword “currency”, the TermsDictionary will add words such as “money”, “cash”, “cheap” and “expensive”. Again, it is the more simple and repeated use of language within these puzzles that allows a parser with less than a full Oxford dictionary to successfully reach a solution. The specialisation of the parser for this task is shown in the limits of the word lists contained in this dictionary.

Alongside this, the Tagger instantiates a **CategoryDictionary** from the puzzle items which splits all multi-word category items into individual words. It ignores any short words, such as “as”, “in” or “the” whilst doing this so as not to unnecessarily bloat the tag pattern. Tagger then considers each clue in turn and each word within that clue in turn. Initially, it creates a string with every tag matched for the length of that sentence.

[so “this..” becomes “A2 To A4…”]

It then condenses this tag pattern down to prepare it for Translator. This involves considering each tag and whether it should remain in the final pattern or not. To achieve this, it makes use of the **PatternBuffer** class. This buffer allows adjacent tags to be considered and kept in if they make a sensible combination, removing anything irrelevant.

[Example]

Tagger then calls upon Translator to turn the finished tagline into relational statements. To do this it compares the part or parts of the line to tag patterns within the **PatternBank** class.

[Example]