The Poetry Pioneer

Nitin Nihalani

Department of Computing

Imperial College London

An interim report for

MEng Computing (AI)

Individual Project

Contents

Chapter 1

Introduction

"Dream in a pragmatic way." - Aldous Huxley

1.1 Motivation

Computational Semantics is a relatively young and fashionable topic in Computational Linguistics. It involves finding representations and algorithms that are able to cope with the *meaning* of linguistic utterances.

Pragmatics is an even younger discipline, concentrating on the context of those utterances. For example, the phrase *"I have a green light"* may mean that I have been granted permission for something, or that I literally have a lamp with a green tint. When we as humans read or listen to any linguistic output, we build a representation of the objects, people, actions, descriptions, relationships and anything else to provide context to the experience.

A similar approach is taken when writing or speaking - one has a purpose and a message that would like to be passed and the language used helps to build such context. For example, if I aim to tell you that I have a lot of tea, I could simply say so or say I am 'drowning' in tea. The latter helps you, the reader, realise that not only do I have a lot of tea but also that I cannot handle it and that I am talking about the drink rather than the leaves.

For a computer to truly converse in a manner indistinguishable to humans, as is the aim of the elusive Turing Test, it must be able to handle pragmatics along with syntax and semantics. This requires a deep understanding of words not as a linguistic unit, but as the objects, actions and descriptions they represent. They should then be used with a purpose when generating text, as human writers and speakers do.

Poetry is a linguistic art form designed to help convey a particular, well-defined message in a memorable and powerful way. Poets write poems that are succinct in length but dense in meaning by employing a number of techniques, such as rhyme and rhythm.

Different types of poetry each have their own sets of constraints, features and priorities that define their best usage. For example, narrative poems convey characters, relationships and actions while descriptive ones are used to give a comprehensive linguistic illustration. Furthermore, rhythm and rhyme can be used to provide melody making it pleasant to hear and applicable to songs, but alliteration can be used to create suspense and a sense of danger.

We aim to create a system that can analyse poetry and build a contextual representation of it, as well as generate poetry with an underlying purpose and message. This process will also act as a catalyst with which to develop computational semantics and pragmatics.

The proposed implementation comes in three phases:

1. First we will write the analysis module, which detects a wide range of poetic features in a single poem. It also aims to represent the context of the poem with Discourse Representation Structures (DRSs), outlined in section 2.4.4.

2. Then we run many poems of the same type through the analysis module and abstract the common features between the given poems into a template. This includes a general structure for the DRSs of that class of poem.

3. Finally we generate poems with a purpose as guided by the structure of the DRS. We will also utilise third-party libraries to build semantically and syntactically valid lines of poetry that also use poetic features. Poeticness and creativity is prioritised in the selection of words and phrases during the generation process. A dry run of this phase is given in the appendix.

1.2 Objectives

The overarching primary objective of this system is *pragmatic competence*. We aim to generate poetry that demonstrates some understanding of context with regards to descriptions, actions and relationships of people and objects, and with careful text and sentence planning for that context.

Thereafter, we wish to create a system that:

- identifies a broad list of features in a single poem.
- abstracts features of a given class of poems or texts that have been analysed.
- learns the features of a wide variety of different classes of poems.
- produces poems, given natural language 'seeds' of inspiration, that are:
  - novel,
  - syntactically valid,
  - semantically interpretable,
  - pragmatically consistent,
  - evident of a set of desired features.
- is able to digress slightly from learned features in its use of poetic techniques in search of creativity.

1.3 Contributions

This project makes contributions towards both Computational Creativity and Computational Linguistics.

- We demonstrate the ability for computer systems to assess written natural language works in terms of its structure, common words and phrases, rhetoric and poetic features such as rhyme, rhythm and alliteration.
- We will investigate the appropriateness of Discourse Representation Theory as a semantic representation of poetry from which we can derive pragmatics in terms of characters, objects and locations, descriptions of them, relationships between them and the actions that they executed.
- We demonstrate the ability to abstract common written features out of a large number of comparable texts.
- We take in a step in the direction of using the web as a source of material and conceptual inspiration for creative acts.

- We demonstrate Discourse Representation Theory as an effective tool for guiding the macro- and micro- planning stages of natural language generation.
- We demonstrate the effectiveness of third-party libraries for the surface realisation stage of Natural Language Generation.
- We provide a tool for poetry creation from natural language seeds of inspiration.
- We investigate the applicability of the new FACE and IDEA descriptive models for evaluation.

Chapter 2

Background

This chapter gives a brief overview of the features and classes of poetry that exist, along with why people are interested in writing them. We will discuss, critique and gather inspiration from the related work in the area of poetry generation that most relate to the approach taken in this paper. We then give brief overviews into the fields of Computational Linguistics and Computational Creativity, both of which are involved in the task of automatic poetry generation. These overviews are by no means complete or comprehensive, but should provide enough information for those not familiar with the areas to understand and appreciate this paper.

2.1 Poetry Theory

To fully comprehend the task that we are about to undertake, we need to have an understanding of poetry as an art form. It is ever-evolving and different styles have emerged over the years. Here we discuss those styles and the underlying reason for why poetry is written.

2.1.1 The Purpose of Poetry

Merriam-Webster dictionary defines poetry as:
'writing that formulates a concentrated imaginative awareness of experience in language chosen and arranged to create a specific emotional response through meaning, sound, and rhythm.'

Let us break this down:

- ***Formulates*** implies that there is method to the process of writing a poem.
- ***Concentrated*** accentuates the fact that poems are generally short, as they are counted in stanzas in lines rather than paragraphs and pages.

- ***Imaginative*** confirms the fact that this is a creative act, that has a level of non-determinism and need not be entirely realistic.
- ***Awareness of Experience*** embodies the need for general background knowledge based on a particular set of experiences that surface when writing a poem.
- ***Language Chosen and Arranged*** reiterates that this is a methodical and systematic art - words are chosen carefully with precise intention.
- ***Create a specific emotional response*** gives the main purpose of the poem - to express a feeling or an idea and elicit emotion in the reader.
- ***Through meaning, sound, and rhythm*** describes that this purpose is not reached purely by words, but other language features.

To summarise, the purpose of poetry is to use one's imagination, knowledge and experience to trigger empathy about a particular subject matter. Poetry is a vehicle through which poets can share a very personal message that they want the reader to experience and remember.

We must keep this in mind throughout the project, as it is important to realise that the features of poetry discussed in the next section are not arbitrary rules on form but purposeful techniques used to make the language more concise and effective.

2.1.2 Features of Poetry

The earlier definition mentions the use of *meaning, sound and rhythm* in poetry. These add an extra layer of subtext to poems to help the author remain concise while still getting the complete message across. We call these techniques *features* of poetry throughout this paper. There are many features of poetry to address, but we have scoped this project down to concentrate on the following common ones.

2.1.2.1 Rhyme

Two words rhyme when they sound similar when spoken out loud. *Cat* and *fat* in figure 2.1 rhyme, as do *mice* and *nice*. Rhyming words need not be spelt similarly, for example, *kite* and *height*.

There once was a big brown ***cat***
That liked to eat a lot of ***mice***
He got all round and ***fat***
Because they tasted so ***nice***

Figure 2.1: A rhyming quatrain often used in teaching poetry

Strict rhyme enforces the exact same sound while weak rhyme only requires that the vowel sounds are the same. Examples of weak rhyme are *turtle* with *purple* and *tragedy* with *strategy*.

A piece of text has a rhyme scheme if there is a pattern of rhyme between its lines. For example, the poem in Figure 2.1 has an *ABAB* rhyme scheme.

Rhyme can also occur within a line (internal rhyme) or between words in the middle of different lines.

Major Purposes

- Pleasant to hear, making the listener feel more comfortable and listen carefully.
- As a mnemonic device.
- Used at the end of lines of poetry and songs making the rhythmic structure more distinct.

2.1.2.2 Rhythm

Rhythm is the pattern of emphasis of syllables that occurs in a line of poetry. There are three major ways of measuring rhythm, often used in tandem - syllabic, quantitative and accentual.

The bartender said
to the neutron, 'For you, sir,
there will be no charge.'

Figure 2.2: A humourous Haiku

Syllabic rhythm enforces a certain number of syllables to be used in a particular line of poetry. Haikus, for example, are three lines long with the first and last lines restricted to 5 syllables and the second to 7. An example is given in Figure 2.2.

Quantitative measures use the fact that some syllables *sound* longer than others when spoken out loud. Long sounding syllables are *stressed* while short ones are *unstressed*.

Accentual measures are similar to Quantitative, but they work on the *tendency to emphasize a particular syllable* when spoken out loud, rather than its length. It is important to note that a word's meaning can change depending on stress. For example, '**ob**ject' is a noun whereas 'ob**ject**' is a verb.

Lines of pre-defined patterns of stressed and unstressed syllables are called *meters*. Lines with meter are made up of individual units called *feet*. The five major foot types in poetry are given in Table 2.1.

Foot Type	Pattern	Example
iamb	unstressed - stressed	des**cribe**
trochee	stressed - unstressed	**po**em
spondee	stressed - stressed	**popcorn**
anapest	unstressed - unstressed - stressed	meta**phor**
dactyl	stressed - unstressed - unstressed	**po**etry

Table 2.1: The major poetic foot types with their corresponding pattern and an illustrative example.

The metre is formed by repeating feet, typically with up to six feet:

- Monometer: 1 foot
- Dimeter: 2 feet
- Trimeter: 3 feet
- Tetrameter: 4 feet
- Pentameter: 5 feet
- Hexameter: 6 feet

All Shakepeare's sonnets are written in iambic pentameter, i.e. five repetitions of unstressed-stressed syllables. The first line of his Sonnet II as an example:
*When **for**ty **win**ters **shall** be**siege** thy **brow**.*

Major Purposes

- Introduces a melody based on the natural intonations of speech.

- Adds a level of predictability and structure that resonates with readers and listeners.
- Emphasizes the message by putting stress on the words that matter.

2.1.2.3 Sound Devices

This project considers four types of sound devices.

The first is **onomatopoeia** - words that imitate or suggest sounds of particular sources. For example, the *pow* of a punch or the *tick-tock* of a clock. This technique has mostly been used in comic books to help the reader experience the sound of the scene to go with the image.

The next three devices are repetitions of a pattern of similar sounds. **Consonance** is the repetition of similar consonant sounds (e.g. *pitter patter* repeats the 'p', 't', and 'r' sounds), while **assonance** is that of vowels (e.g. *doom and gloom* repeats the 'oo' sound). **Alliteration** is a special case where the repeated sound occurs at the beginning of consecutive words. *Zany zebras zigzagged through the zoo* has alliteration on the letter 'z'.

Major Purposes

- Poets use onomatopoeia to help describe actions or atmosphere richly. A famous example is the nursery rhyme 'Old MacDonald', which uses onomatopoeia of the sounds that animals make to describe the farm, figuratively placing the reader or listener in the farm itself.
- Alliteration, consonance and assonance are pleasant to listen to when spoken out loud.
- Can be used to add drama to an action.
- Sometimes used to suggest danger.

2.1.2.4 Structure

The structure of the poem is the organisation of its lines in a poem. The main unit is the *stanza*, which is a fixed number of *lines* grouped by rhythmical pattern.

There are four major types of stanza:

- Couplet: 2 lines
- Tercet: 3 lines
- Quatrain: 4 lines
- Cinquain: 5 lines

Stanzas can also be called *verses*, which have the added property of a rhyme scheme. A *chorus* is a special type of verse that is repeated throughout a poem.

Features of the structure of a poem include:

- The number of stanzas.
- The number of lines per stanza.
- The number and positions of repeated lines.
- The number and positions of repeated stanzas.
- Enjambment; the continuation of a sentence over a line-break

The Haiku in Figure 2.2 has a single tercet structure with no repetitions. Songs are generally several stanzas long, with a chorus interleaving longer non-repeating verses.

Major Purposes

- Helps to guide the reader through the story.
- Forces the poet to be more succinct and purposeful.
- Manages the storyline - changes in stanza often suggest a change in perspective or message.
- Repetition helps drive home the main message.
- Ties several thoughts together into one continuous flow.

2.1.2.5 Symbolism and Imagery

Symbolism and imagery are general terms for creating an overall image in the reader's mind by describing a subject or object as something else with desired qualities.

Techniques include:

- **Metaphor**: an object is described as another object with a set of desirable characteristics. For example, saying someone is a lion immediately creates the image of bravery, intimidation and power.
- **Simile**: an object or action is specifically described using an adjective or adverb, but compared to another object that is a stereotypical example of that description. The phrases 'like a' and 'as a' are often used, e.g. *Runs like a cheetah*, *Slippery as an eel*.
- **Hyperbole**: unrealistic exaggeration, often used in tandem with metaphor e.g. *Cried a river of tears*.
- **Powerful Verb**: a more exciting way to describe an action using unusual verbs, e.g. *Wormed through the crowd*.
- **Personification**: using actions and properties associated with sentient objects to describe inanimate ones. Explained further in section 2.1.2.6
- **Onomatopoeia**: as explained in section 2.1.2.3, using imitations of known sounds to richly describe actions and atmosphere.

Major Purposes

- Explain complex concepts concisely.
- Induce empathy from the reader by relating it to something they understand.

2.1.2.6 Context and Personification

Poetry is similar to storytelling in that it has persona around whom the poem is written. Understanding who or what they are, their descriptions and their actions are all part of the underlying message that the poet wants to get across.

Personification is a technique used by poets to give inanimate objects life, expressing actions and descriptions as if it were sentient. This is a powerful technique that relates to imagery, helping poets make abstract messages clearer. For example, *the moon smiled* gives the moon life by describing it as having performed a sentient action with full intention of doing so. Noting the use of personification can make the context of the poem clearer, as inanimate objects are often the subject of the poem.

Major Purposes
Context is the underlying message in its bare form. It is the story that the poet wishes to tell and guides the use of all other features.

In this paper, we aim to extract characters and differentiate them by their descriptions and actions. This is vital in understanding the poem and can help us generalise the uses of features when attempting to produce a coherent story as the backbone to the generated poem. Furthermore, it will help determine the type of poem (narrative, lyrical, descriptive etc.) and will help guide generation of poems of a particular type.

2.1.3 Classification of Poetry

We define a type of poetry as a particular form of poem with a set of unique features, including those described in the previous section. Some types are very popular and have had their styles, features and purposes documented and taught. Out of these grew categories of different types that tend to be used for similar purposes.

This project attempts to derive these categories and some popular types of poetry by analysing many comparable poems.

2.1.3.1 Categories

There are many types of poem all with different form. However, there are only three main categories of purpose for a poem:

1. *Lyrical* poems have an identifiable speaker whose thoughts and emotions are being expressed in the poem. This means that poems of this category have very few characters, a song-like structure and tend to be in a reflective tone, generally using a lot of symbolism. Maya Angelou's *I Know Why The Caged Bird Sings* is an example of this, along with many songs.

2. *Descriptive* poems describe the surroundings of the speaker. This is identifiable by the use of adjectives and complex imagery. Many objects may appear in this type of poem to be able to give an in-depth description of the environment and atmosphere. There will be very few action verbs used.

3. *Narrative* poems concentrate on telling a story. It therefore has a coherent plot line, several characters with explicit relationships between them, action and climax. Ballads and Epics are types of narrative poems.

Some popular poem types do not fall under any one bracket as they can be used in any of the above categories. Examples include Haikus and Limericks.

2.1.3.2 Popular Types

As well as determining the category of poems, we aim to be able to detect and reproduce some popular types of poetry. For this project, we will concentrate on:

- *Haiku:* single tercet structure with 5-7-5 syllabic rhythm.
- *Limerick:* single cinquain structure with AABBA rhyme scheme. Lines 1, 2 and 5 have 7-10 syllables, while lines 3 and 4 have 5-7 syllables. The first line tends to begin with "There was a..." and ending with a person or location. Limericks are usually used for humour as the last line is generally a punchline.
- *Sonnet:* 14 lines, each in iambic pentameter with an ABAB CDCD EFEF GG rhyme scheme, i.e. three quatrains followed by a rhyming couplet.
- *Elegy:* usually used to mourn the dead, its lines alternates between dactylic hexameter and pentameter in rhythm. It has no particular rhyme scheme, although does still use rhyme.
- *Ode:* Description of a particular person or thing, using plenty of similes, metaphors and hyperbole.
- *Ballad:* Tells a story and has a number of quatrains, each with an AABB rhyme scheme. Lines alternate between iambic tetrameter and iambic trimeter.

- *Cinquain:* as the name suggests, this has 5 lines. They are not rhymed, but have a 2-4-6-8-2 syllabic pattern.
- *Riddle:* Riddles describe things without telling what it is, using anaphora to refer to it. Ususally told in a number of rhyming couplets.
- *Free Verse:* No particular features attached to this type.

2.2 Lessons from Related Work

This section looks at six important previous attempts at automatic poetry generation. They each have some aspect of investigation or experimentation that have influenced this project. Conversely, each of these attempts has its limitations that we look to overcome in this project.

2.2.1 Actively Gather Inspiration

Colton et al. published a paper in the International Conference of Computational Creativity 2012[17], whose main objective was to describe the first poetry generation system that satisfied the FACE Descriptive model[18]. It is a *Form Aware*[35] implementation that constructs templates of poems based on constraints of poetic features.

The most interesting point of this paper was its admission that inspiration cannot come from the technology and must come from the user. By taking this into account, it now takes inspiration from news articles as seen in Figure 2.3. However, since its objective was focused on passing a particular evaluation model, the poems created by this system are relatively simple and the processes rudimentary - using randomness rather than semantic applicability in word selection.

2.2.2 Constrain to Improve Creativity

Recently, Toivanen et al. attempted a solution that used off-the-shelf constraint solvers[43] to produce poetry. Their solution, illustrated in figure 2.4, also received inspiration from external sources. This is used to build the set of candidate words, form requirements and content requirements that are passed into a constraint solver

It was generally a bad news day. I read an article in the Guardian entitled: "Police investigate alleged race hate crime in Rochdale". Apparently, "Stringer-Prince, 17, has undergone surgery following the attack on Saturday in which his skull, eye sockets and cheekbone were fractured" and "This was a completely unprovoked and relentless attack that has left both victims shocked by their ordeal". I decided to focus on mood and lyricism, with an emphasis on syllables and matching line lengths, with very occasional rhyming. I like how words like attack and snake sound together. I wrote this poem.

Relentless attack
a glacier-relentless attack
the wild unprovoked attack of a snake
the wild relentless attack of a snake
a relentless attack, like a glacier
the high-level function of eye sockets
a relentless attack, like a machine
the low-level role of eye sockets
a relentless attack, like the tick of a machine
the high-level role of eye sockets
a relentless attack, like a bloodhound

Figure 2.3: The Guardian article used for inspiration(left) and the resulting poem(right).

with a manually encoded static constraint library powered by Answer Set Programming.

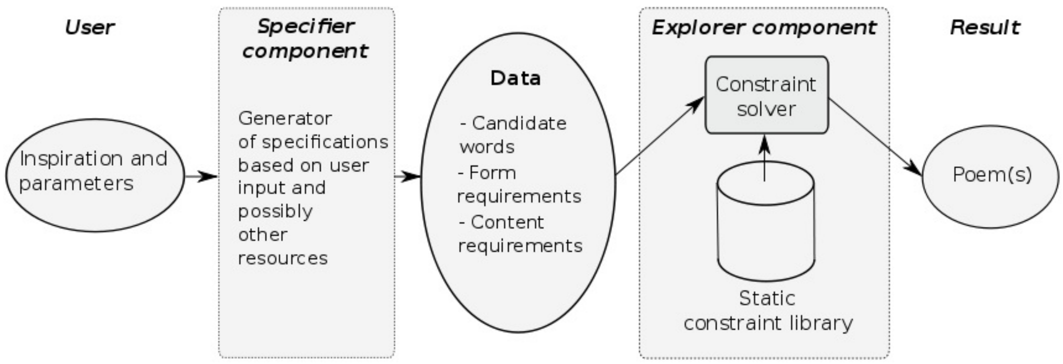


Figure 2.4: Complete poetry composition workflow.

The idea that constraints do not hinder but rather help the creative process is an attractive one for Computational Creativity research. Constraining words and other requirements for each particular word position is a natural technique for constraint programming, but extremely restrictive. First, the size of each line must be defined by number of words *and then* by rhythm and other poetic features. Secondly, once the candidate words are chosen there is no scope for further filtering. Finally, the

N SG VB, N SG VB, N SG VB! PR PS ADJ N PL ADJ PRE PR PS N SG: – C ADV, ADV ADV DT N SG PR VB! DT N SG PRE DT N PL PRE N SG!	*Music swells, accent practises, theatre hears!* *Her delighted epiphanies bent in her universe:* *– And then, singing directly a universe she disappears!* *An anthem in the judgements after verse!*

Figure 2.5: The POS template used for constraint input(left) and the resulting poem(right).

structure of the poem in terms of its parts-of-speech (section 2.4.2) tags must be defined beforehand and is taken from previous poems of the same type, as seen in Figure 2.5. Even though this is an efficient method that has produced impressive results, it is too restricted to produce truly creative work.

2.2.3 Learn from Experience

Ray Kurtzweil Cybernetic Poet (RKCP), created by Kurtzweil himself[33], addresses the issue of having a predefined template. He uses a stochastic approach that utilises of n-grams to build lines from words. The system was trained on a selection of poems that created a template and n-gram corpus from those poems. RKCP would use this to create similar types of poems. Some heuristics were employed to ensure that poems were not exact copies of other poems and to maintain a coherent theme.

Scattered sandals
a call back to myself,
so hollow I would echo.

Figure 2.6: A haiku written by Ray Kurzweil's Cybernetic Poet after reading poems by Kimberly McLauchlin and Ray Kurzweil

This method is more flexible and has granular word selection. However, the vocabulary would still be limited and the form of the poem is not well defined due to being probabilistic. We can see that in Figure 2.6, the attempted Haiku has a syllabic rhythm is 4-6-7 as opposed to the required 5-7-5. A specific purpose or storyline is not definable and the use of imagery is only probabilistic. A lot also depends on the poems in the corpus, limiting semantic capabilities and word selection quality.

2.2.4 Choose Words Carefully

MCGONAGALL[35] takes a semantic representation of a sentence, called *semantic expressions*, as input into an NLG system. For example, the semantic expression of *"John loves Mary"* would be {*john(j), mary(m), love(l, j, m)*}

These are used as starting points for initialisation of his evolutionary system that uses stochastic methods to determine the best values to be carried forward to further iterations.

They play. An expense is a waist.
A lion, he dwells in a dish.
He dwells in a skin.
A sensitive child,
he dwells in a child with a fish.

Figure 2.7: Resulting MCGONNAGAL poem when seeded with a couple of lines of Hilaire Belloc.

Of particular note is the structure of a lexical entry into the system. It is enriched with much semantic information, as in Figure 2.8, that backs up the fitness score and helps MCGONAGALL form syntactically and semantically correct sentences. We will use much of his ideas in this area. However, contextual coherence is lacking because of the restrictions imposed on evolution. It does not take particular types of poetry into account and there is little scope for creativity due to the strictness of grammar generated.

2.2.5 Derive Insight from Worldly Knowledge

Tony Veale's daring approach to knowledge-based poetry generation[44] concentrates on symbolism and imagery - arguably the hardest tasks in automatic poetry generation. He uses norms and stereotypes to build a structure that uses various words to describe objects and derive stereotypical characteristics. Out of this grew a very useful tool - Metaphor Magnet[45], which was used to create the impressive poetry shown in Figure 2.9.

His methods have obvious limitations in that they do not consider rhyme, rhythm or any other poetic feature other than symbolism. However, we will take advantage

Field	Value
Key	`lion_n`
Orthographic spelling	*lion*
Phonetic spelling	`[L,AY1,AH0,N]`
Semantic expression	lion(X,Y)
Semantic signature	$\boxed{X,Y}$
Anchored trees	`I_N_N, I_C_NP, A_R_C_NP`
Feature structure	$\begin{bmatrix} \text{CAT} & n \\ \text{AGR} & \begin{bmatrix} \text{NUM} & sg \\ \text{PERS} & 3 \\ \text{3RDSG} & + \end{bmatrix} \\ \text{PRON} & - \\ \text{PROP} & - \\ \text{SUB} & \begin{bmatrix} \text{ANIM} & + \end{bmatrix} \end{bmatrix}$

Figure 2.8: Semantically enriched lexical entry for *lion* in MCGONNAGAL

inspiration from the idea of using norms and stereotypes to give this system more symbolic choices of words and phrases.

2.2.6 Dare to be Different

WASP is one of the first attempts at an automatic poetry generator. It is a rule based system that takes a set of words, a set of verse patterns and returns a set of verses[28]. It uses heuristics to guide the construction to fit structure, but no semantic limitations are enforced.

This has obvious limitations but Gervas, the creator of this system, does make a good

My marriage is an emotional prison
Barred visitors do marriages allow
The most unitary collective scarcely organizes so much
Intimidate me with the official regulation of your prison
Let your sexual degradation charm me
Did ever an offender go to a more oppressive prison?
You confine me as securely as any locked prison cell
Does any prison punish more harshly than this marriage?
You punish me with your harsh security
The most isolated prisons inflict the most difficult hardships
O Marriage, you disgust me with your undesirable security

Figure 2.9: 'The legalized regime of this marriage', a poetic view of marriage as a prison

point that poetry's creativeness is somewhat down to daringness of transgression. We keep this in mind to allow some level of randomness and mutation from expected norms in this project.

2.3 Brief Overview of Computational Creativity

Simon Colton and Geraint Wiggins define research in this area as:
The philosophy, science and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem to be creative.[19]

In the context of automatic poetry generation, we are creating a system that *takes on the responsibility* of generating aesthetically pleasing, meaningful and novel poems. The poems still need to be sufficiently similar to existing works created by humans such that it *exhibits behaviour* to which *unbiased observers* can relate and recognise.

This definition has evolved from one where behaviour was *deemed creative if and only if it can be exhibited by humans*[47]. However, recent developments in the area have lead to the requirement of more quantitative measures for evaluation than Turing-style tests, such as the FACE and IDEA descriptive models[18].

This area of research has come under scrutiny for philosophical reasons, but has had support from Alan Turing and other pioneers of Artificial Intelligence. It has since been accepted as a valid area of research, with the annual International Conference on

Computational Creativity heading into its fifth year.

Successes of Computational Creativity:

- Simon Colton's *Painting Fool*[16] produced paintings that managed to trick art lovers into believing that it was the work of a talented human artist. An example is given in Figure 2.10.
- *JAPE*[10], created by Ritchie and Binsted in 1994 was given a general, non-humorous lexicon and generated puns as answers to questions. For example:
  Q:What do you call a strange market?
  A: A bizarre bazaar.
- *Iamus* by Gustavo Diaz-Jerez[23], which composed music entirely on its own that was then recorded by London Symphony Orchestra.
- *The Policeman's Beard is Half Constructed*[13] is recognised for being the first book, which included some poetry, to have been written entirely by a computer program, RACTER.

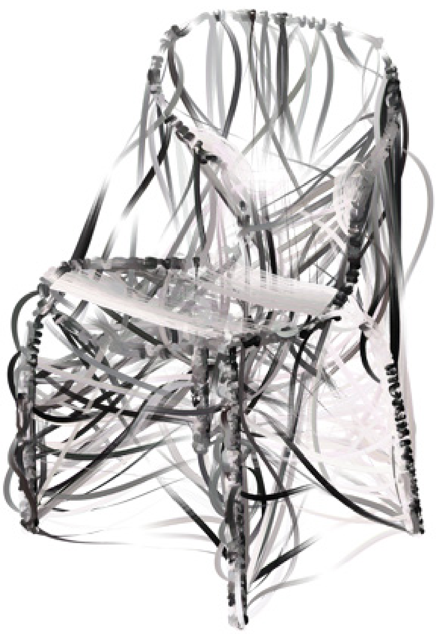


Figure 2.10: Chair #17 at the Performing Sciences Exhibition, La Maison Rouge, Paris, Sept 2011

2.4 Brief Overview of Computational Linguistics

Computational Linguistics is a wide area of research, covering Speech Recognition, Natural Language Processing and Generation and with overlaps in several other areas such as Machine Learning and Knowledge Representation. In fact, Daniel Jurafsky and James H. Martin needed almost a thousand pages to cover the foundations of this area[29].

Automatic poetry generation borrows many techniques and terminology from Computational Linguistics. Here we will briefly discuss the major ones in general and in the context of machine poetry analysis and generation. For an in depth general study of Computational Linguistics, we refer the interested reader to Jurafsky and Martin's book.

2.4.1 Words

Words are the fundamental building blocks of language. They have been studied for the creation of spell-checkers, text-to-speech synthesis and automatic speech recognition. Two major subsets where poetry is concerned is the study of pronunciation and morphology.

The CMU Pronunciation Dictionary[46] has taken steps towards computationally modelling the phonetics of words, using the ARPAbet phoneme set (see Table A.2 in the Appendix). It is highly important for poetry generation as it helps machines reason about rhyme and sound devices by simply comparing phonemes. It has over 133,000 words mapped to corresponding pronunciations.

To illustrate how this works, let us take two words that are spelled differently but pronounced the same - *kite* and *height*. The Jaro-Winkler distance, a normalised score of similarity between strings, for the tail of these words (in search of rhyme) gives 51.11%, indicating that it is barely probable that they rhyme if we only looked at spelling. Their corresponding phoneme sets are *'K AY1 T'* and *'HH AY1 T* respectively. Now it is trivial to compare them computationally and see that the tails are exactly the same and the words therefore rhyme.

Notice the '1' appended to the 'AY' phonemes. Vowel phonemes come with a digit appended to them that defines the emphasis placed on this syllable:

- 0: unstressed
- 1: stressed
- 2: light/secondary stress

Morphology of words is the study of putting words together with *morphemes*, the smallest unit of grammar. To use Jarufsky and Martin's example, the word *fox* consists of a single morpheme that is itself, but *cats* has two morphemes, *cat* and *-s*. This is of vital importance in our project as we need to understand the difference between different forms of the same word and how they relate to context. Furthermore, when generating text we wish to produce coherent grammar with consistent tense and perspective.

The CLiPS Pattern library has a number of tools for morphology of words. It provides a method of changing a word into its first, second or third person version, pluralisation and finding superlatives.[22]

SimpleNLG

2.4.2 Syntax

Syntax is the glue that binds words together. It gives us an understanding of the grammatical relationship between words and guides the building of phrases and sentences.

Core to this area of research is *part-of-speech (POS)* analysis, which provides a model for grouping words together correctly, taking into account how words depend on each other. The big success story in this area is The Penn Treebank Tagset, an enormous corpus of annotated POS information [36]. The full tagset is given in Table A.3 in the Appendix. This accelerated progress of research in the area, as the paper had expected.

From these POS tags, we are able to create *grammars*, the structural rules of phrases and sentences, and *parsers* for those grammars that are able to extract grammatical structure from unstructured text.

For example, the phrase *John loves Mary* would be represented as in Figure 2.11 if parsed with a grammar based on The Penn Treebank tagset.

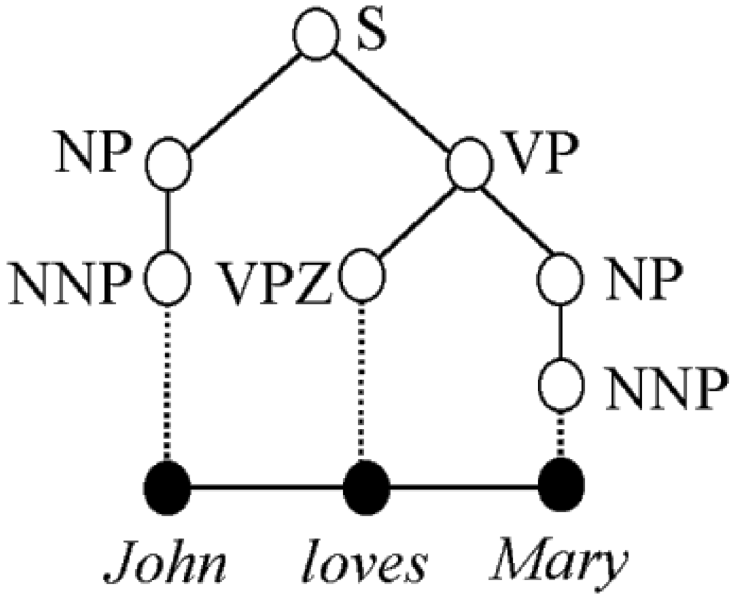


Figure 2.11: Parse tree of *'John loves Mary'*

Python's Natural Language Toolkit (NLTK)[11] is a suite of text processing libraries, corpora and lexical resources that is heavily used in this project, particularly for syntactical purposes. It will allow us to use The Penn Treebank tags as well as produce our own grammar and parser that can be used to parse most poetry. This is a challenge because we cannot expect poetry to follow grammatical rules as strictly as discourse. However, the pay-off will be that we can model the context of the poem, leading to better analysis of semantics and pragmatics of poetry.

2.4.3 Semantics and Pragmatics

Chomsky used the famous example *'Colourless green ideas sleep furiously'* to show that a valid grammatical syntax can be completely nonsensical[15]. This illustrates the importance of the study of semantics, the meaning of words and phrases, as well as pragmatics, the way context affects semantics. Suppose the context around Chomsky's example was a person with old (black-and-white, colourless) ideas of making money (green) that he wants to bring back (was put to sleep, but not peacefully), then this line would make some sense in a poetic way.

This is yet another major challenge for poetry generation. Poems are concise but have many layers of meaning that are subtle even to human readers and that only the best of poets can fully control. The different categories of poems each have their own pragmatics and there is always a fundamental message and purpose supporting these layers meaning. Veale argues that 'poetic licence' is not a licence but a contract that allows a speaker to take liberties in language in exchange for real insight[44]. We must be careful not to abuse this by slewing unrelated words together and expecting the reader to do the work of finding meaning and context.

Various methods of understanding semantics in natural language have been proposed. A very popular one is First Order Logic. In particular, Hans Kamp proposes Discourse Representation Theory (DRT) as a way of modelling language in such a way. We use this theory in this project and will go into more depth in the next section.

Johan Bos and his team have begun the Groningen Meaning Bank (GMB) project[8], a large semantically annotated corpus in lieu of The Penn Treebank, in the attempt to bring the same success and acceleration to this sub-field of research. They use DRT as the backbone to an assembly of third-party tools to annotate semantics, as can be seen in Figure 2.12.

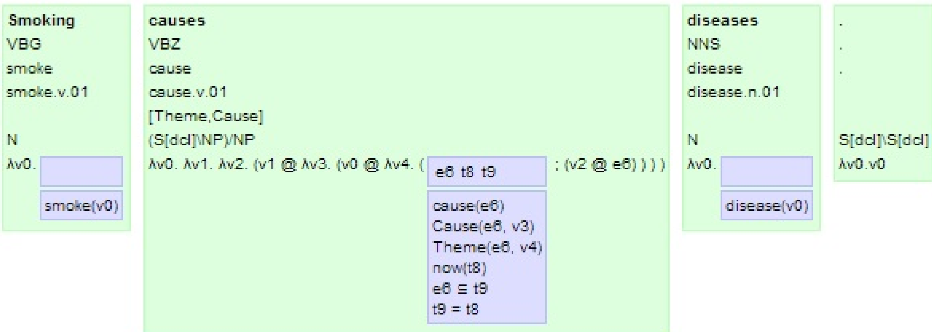


Figure 2.12: Semantic structure of the sentence *Smoking causes diseases.*

However, the GMB project is still in early stages and has only annotated open license news articles up until now, which is not a suitable corpus for poetry generation. However, we will support the coordination of third-party tools and will consider using the following to boost the semantic and pragmatic skill of our poetry generator, particularly with regards to the problem of producing vivid imagery and pragmatic parsing.

- *WordNet*: a lexical database that provides hierarchical, conceptual-semantic and lexical relations of 155,287 English words.[38]

- *VerbNet*: a lexical database that groups verbs by semantic and syntactic linking behaviour.[42]
- *FrameNet*: a lexicon with framing semantics to define and constrain the building of clauses around individual words.[6]
- *ACE*: a classification of names, places and other proper nouns for Named Entity Recognition.[25]
- *PropBank*: an annotated corpus a million words defining and providing argument role labels for verbs[31]
- *NomBank*: similar to PropBank, but for nouns instead of verbs.[37]
- *Wordnik*: a multi-faceted dictionary and semantic language database collected from a variety of sources[1]
- *Metaphor Magnet*: a web application that maps commonplace metaphors in everyday texts[45]
- *Oxford Collocations Dictionary*: a source of word pairings and phrases that occur with greater than chance probability[20]
- *University of South Florida Free Association Norms*, a database of various types of word associations[39]
- *LinguaTools DISCO*: a tool to derive semantic similarity between words based on statistical analysis of large text collections[32]
- *ConceptNet 5*: a semantic general knowledge network[34]
- *Written Sound*: a dictionary of onomatopoeia to their meanings and associated objects[2]

Not all of these tools will be used in this project due to the limited scope. The selected few will be justified in the implementation section with regards to meeting the objectives of this paper. Others will be suggested for future extensions to this project.

2.4.4 Discourse Representation Theory

Discourse Representation Theory (DRT)[30] is a framework for investigating semantics of natural language. DRT is becoming the accepted theory of meaning representation[8][12] and is fundamental to the task of semantic modelling of natural language.

Abstract mental representations of DRT are Discourse Representation Structures (DRSs). It is designed to be able to combine meaning across sentences and cope with anaphora (e.g. pronouns in place of nouns).

Using Kamp's example, if we take the sentence *A farmer owns a donkey* and convert it into a DRS, we get the following notation:
{[x,y: farmer(x), donkey(y), own(x,y)]}

If we then say *He beats it.*, it will produce:
{[x,y,z,w: farmer(x), donkey(y), own(x,y), PRO(z), PRO(w), beat(z,w)]}.

We can then use anaphora resolution on this DRS to produce:
{[x,y: farmer(x), donkey(y), own(x,y), beat(x,y)]}

We can see that this is similar to the notation used by Manurung in MCGONNAGAL, described in section 2.2.4.

This method has evolved over the years to take tense and aspect into account, providing temporal reasoning in natural language sentences. Accuracy of anaphora and presupposition (e.g. saying 'animal' instead of 'cat') resolution has improved with the use of the third party tools mentioned earlier in combination with the ideas of Blackburn and Bos[12].

Extending this example, we may wish to model the sentence *Every farmer who owns a donkey beats it.*. DRT provides an elegant solution for this using first order logic style 'for all':
{[x][y][farmer(x), donkey(y), own(x,y) ->beat(x,y)]}

This allows us to provide background knowledge to the system and make inferences on it. As a result of its usefulness in many applications, NLTK has included DRS manipulation and anaphora resolution into its core 'sem' package.

In poetry, like any text, we can use this to analyse the semantics and pragmatics of the storyline. Furthermore, we hypothesis that it can be used to model each poetry type since they each have their own forms and purposes, and provide coherency to the story when generating text.

2.4.5 Anaphora Resolution and Presupposition Projection

Need some stuff here. Probably remove the DRT section as well.

2.4.6 Natural Language Generation

Natural Language Generation is the term for putting some non-linguistic form of content into understandable text in a human language. Reiter and Dale give the framework[41] illustrated in figure 2.13 for the process of generating natural language.

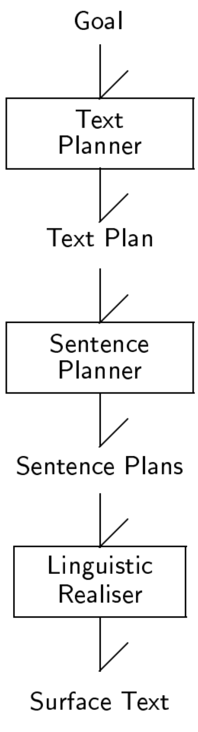


Figure 2.13: Reiter and Dale Natural Language Generation process

A similar model was propsed by Bateman and Zock[9], which includes four stages:

1. *Macro Planning:* Overall content of the text is structured.
2. *Micro Planning:* Specific words and expressions are decided.

3. *Surface Realisation:* Grammatical constructs and order are selected.

4. *Physical Presentation:* Final articulated text is presented.

These are theories upon which natural language generation tools, such as simpleNLG[27], have been designed. There are others, such as grammar generation implemented by NLTK (similar to the one used in MCGONNAGAL) or the constraint programming technique used by Toivanen et al. explained in section 2.2.2, but we find that this process is not granular enough, jumping from the goal to the surface text without enough consideration for the individual words used.

The inputs into this system can vary. One popular problem is raw numerical data into a textual summary. In our case, however, we look at symbolic representations of content in the form of a DRS. Basile and Bos argue that DRSs are appropriate for all stages of the Bateman and Zock example[7]. Gardent and Kow have also proposed a symbolic approach to the Surface Realisation phase, working from first-order logic as the input form[26].

In this paper, we will look at using DRSs to guide the content, i.e. the Macro Planning stage. However, we feel it is restrictive to use it for Micro Planning, since it gives no scope the use of symbolic imagery, or Surface Realisation, as it forces grammatical perfection.

Chapter 3

Poem Analysis

The first phase of implementation involves writing a suite of algorithms to analyse a single poem in terms of the features mentioned in section 2.1.2. The aim is to run a large collection of poems through this analysis to learn the usage patterns of poetic features for any collection of poems. This is in line with the ultimate aim of generating poems without hard-coded rules for different types of poems.

The algorithms will cover the detection of the use of:

- Rhyme and internal rhyme
- Rhythm, including meter and syllable count
- Alliteration, assonance, consonance, onomatopoeia and other sound devices
- Structure, tense, point of view and repetition

Other algorithms will attempt to understand the context of the poem to extract:

- Characters
- Objects
- Locations
- Descriptions
- Relationships
- Actions
- Symbolism including metaphors, similes and personification

The output of this phase is a full analysis of a single poem. In lieu of this, we will walk through the implementations of each of these algorithms using the poems in Figures 3.1 and 3.2 as case studies.

There once was a big brown cat
That liked to eat a lot of mice.
He got all round and fat
Because they tasted so nice.

Figure 3.1: A rhyming quatrain often used in teaching poetry

The limerick packs laughs anatomical
Into space that is quite economical.
But the good ones I've seen,
So seldom are clean,
And the clean ones so seldom are comical.

Figure 3.2: A limerick about limericks

[['DH', 'EH1', 'R']]
[['W', 'AH1', 'N', 'S']]
[['W', 'AA1', 'Z'], ['W', 'AH1', 'Z'], ['W', 'AH0', 'Z'], ['W', 'AO1', 'Z']]
[['AH0'], ['EY1']]
[['B', 'IH1', 'G']]
[['B', 'R', 'AW1', 'N']]
[['K', 'AE1', 'T']]

Figure 3.3: The different ways of pronouncing the first line of the cat poem

3.1 Obtaining Phonetic Structure

Poets choose words based the *sound* when spoken out loud as well as their (literal or symbolic) meaning. As explained in section 2.4.1, we can use Carnegie Mellon University Pronounciation Dictionary (CMUPD) to get around the difficulty of determining phonetic structure. A word in the CMUPD is mapped to a list of different pronunciations for the same word. Each pronunciation is a list of phonemes that make up that particular pronunciation of that word, including indication of emphasis on the syllables.

We want to convert the poems into their phoneme lists for use by the detection algorithms. This needs to be done word by word, so we first need to *tokenise* the sentence. Tokenisation involves splitting each sentence into a list of its basic components; words and punctuation. Once we have done that, we can simply iterate through the list and run each word through the CMUPD. Some words have multiple pronunciations, so we consider each possible permutation of pronouncing each line of the poem. Each of the pronunciations of the first line of Figure 3.1 is shown in Figure 3.3.

Unfortunately, the CMUPD only has about 133,000 words. This means that we are occasionally unable to translate to the phonetic structure, particularly in Shakespearean poems. We get around this by temporarily converting the word into its closest match that exists in the dictionary and returning the phonetic structure in its place.

Python *difflib*[3] provides a function to find the closest matches of a word to a list of words, based on the Ratcliff/Obershelp pattern recognition algorithm. The complexity of this algorithm is quadratic in the worst and average case, linear in the best case.

The behaviour is based on how many subsequences the words have in common. Since we are only dealing with single words at a time, we have a higher chance of the best case. On average it takes 0.81 seconds to look up a 4-letter word and 1.09 seconds to look up an 10-letter word using machine specifications given in Appendix A.4, which is fast enough given that this is a pre-processing phase, not a time-sensitive one.

We could use a variety of other techniques to get around this problem other than string matching:

- Break many syllable words into likely part-words, e.g. *'thrift'* and *'less'* instead of *'thriftless'*
- Try all combinations of stress and syllables
- Train a finite-state transducer model as in Dobrivsek et al. 2010[24]

The first option only works in a limited number of situations, most of which are handled by the string matching solution. For example, *'thriftless'* would become *'shiftless'*, which has an identical phonetic structure. The second option can result in poor performance and more false negatives or false positives than would be worth the added processing.

The final option is the most viable and would be used if the generation phase depended on perfect readings of the poems, such as in the stochastic n-gram methodology of RKCP (section 2.2.3). However, as we only use this as an approximation for a more general solution in later phases, such precision is beyond the scope of this project.

3.2 Rhyme

We want to detect end-line and internal rhyme, as described in section 2.1.2.1. Along with detecting it, we want to be able to build a normalised rhyme scheme representation for easy analysis in the abstraction phase.

First we collect the phoneme set of the words for which we wish to build a rhyme scheme. For end-line, we collect the last word of each line of the poem. For internal rhyme, we collect the all the words in a particular stanza.

Once we have our list, we can run it through Algorithm BLAH.

```
1 for each word in the list:
2 for each pronunciation of the word:
3 for each phoneme in the pronunciation:
4 if this phoneme is stressed:
5 get the tail of the pronunciation from this phoneme onwards
6 get the rhyme phoneme pattern of this tail
7 if we have seen this pattern before:
8 assign it the corresponding rhyme token
9 else:
10 assign this pattern a new rhyme token
11
12 append this token to a set of possible tokens for this word
13 append the set of possible tokens for this word to the unzipped rhyme scheme
14 build all possible permutations of rhyme scheme from they unzipped rhyme scheme
15 normalise the rhyme schemes
```

There are a few tricks to this algorithm, namely obtaining the rhyme phoneme pattern on line 6 and the process of building and normalising the rhyme scheme in lines 12 to 15.

3.2.1 Obtaining the Rhyme Phoneme Pattern

Two words rhyme when:

- the last phoneme of each word match
  and
- the vowel sounds after and including the first stressed syllable match **in order**

The algorithm finds the first stressed syllable in line 4. At that point we only need to iterate through the rest of the pronunciation and check the vowel sounds and the last phonemes. Putting them together, in order, gives the unique rhyme phoneme pattern.

For EXAMPLE strategy and tragedy, kite and height - colourful diagrams!

3.2.2 Building and Normalising the Rhyme Scheme

Each unique rhyme phoneme pattern is represented by a single capital letter starting with 'A'. This is the standard convention for rhyme schemes used in theory.

The complication comes in the various ways of pronouncing a single word. Therefore, we create a list for every possible rhyme pattern for a each word. The list of these lists is what the algorithm refers to as the 'unzipped' rhyme scheme on line 12 and 13. Figure BLAH shows the unzipped rhyme scheme for the limerick in Figure BLAH. Note that the word *'anatomical'* has two pronunciations that affect the rhyme pattern, according to the CMUPD.

FIGURE OF THE UNZIPPED RHYME SCHEME

We then build every possible permutation of this list, which then gives us all of the possible rhyme schemes for the given words. We leave this list as it is and do not make a claim for any rhyme scheme to be more likely than any other at this stage.

EXAMPLE RHYME SCHEMES OF CAT AND LIMERICK

3.3 Rhythm

We attempt to recognise all three types of rhythm described in 2.1.2.2.

3.3.1 Detecting Syllabic Rhythm

The number of syllables in a word is equal to the number of vowel phonemes it has. We know that all vowel phonemes have a stress marker appended to them so we can just count the number of stress markers in the word.

However, Syllabic Rhythm is done on a line-by-line basis. Therefore we tokenise each line and aggregate the syllables across all words in the line.

If we take one permutation of the line in Figure 3.3, we can see that each word has one vowel phoneme, so each word in that sentence is monosyllabic. Therefore the number of syllables in the line is 7.

The full syllabic rhythm for the poems in Figure 3.1 and 3.2 are 7-8-6-7 and 11-6-5-11-11 respectively. We represent these as a list of integers and create a separate list for each stanza.

3.3.2 Detecting Quantitative and Accentual Rhythm

While theory counts rhythm in terms of metre and feet, as described in 2.1.2.2, some poems might have rhythm without following one of these pre-defined popular styles. Instead of searching for them specifically, we will extract the pattern of stressed and unstressed syllables for each line without making a claim on how it matches to theory yet.

We choose to do this mainly because there could be multiple possibilities for the stress pattern of a line due to variations in emphasis for particular words, e.g. _'**ob**ject'_ and _'ob**ject**'_. The variability of possible stress patterns is compounded by some limitations of the CMUPD:

- Some words are restricted to a single stress pattern even though it does not lose or change the meaning of the word if it was emphasised differently. For example, _'**quan**tity'_, _'quan**ti**ty'_ and _'quanti**ty**'_ are all valid, but the CMUPD only recognises the first because the other two are unusual in normal speech (but not for poetry).
- Similarly, the CMUPD fixes monosyllabic words to one of stressed or unstressed as in Figure 3.3. However, they can be either so we need to allow for both possibilities.
- Stresses on some words have changed over time. For example _**prov**ed_ was pronounced _prov**ed**_ in Shakespeare's era.

- Though not necessarily a limitation, we do not recognise light or secondary stress in a word (the '2' stress marker). We therefore assume it could be either '1' or '0'.

- Poetic license often allows words to be pronounced with an entire extra syllable. For example *'de-served'* could be pronounced as *'de-ser-ved'*.

We account for these by increasing the number of possible phonetic stress marker readings. However, this will lead to an unrealistically large number of possibilities. For example the line in Figure 3.3 will give us any combination of '1's and '0's since they are all monosyllabic and by the second point above we want to allow for either.

We narrow this down by taking away any occurrences of the same stress three or more times in a row in the same line, because no one will ever read a poem in this way. Like with rhyme, we do not make a claim for any stress pattern to be more likely than any other at this stage.

FIGURE of possible stress patterns for first line of cat and limerick

3.4 Sound Devices

We aim to detect each of the sound devices described in 2.1.2.3, alliteration; assonance and consonance.

The implementation for all three of them is very similar. For each line, we map each phoneme to the number of times it occurs and return the phonemes that occur more than once.

The only difference is that assonance only looks at vowel phonemes (i.e. with a stress marker) and consonance only looks at non-vowel phonemes. Alliteration only looks at either the first phoneme of each word in the line, or the first stressed phoneme and the phoneme immediately before that.

For example, take the phrase *"Slither slather"*:

- **Consonance**: The phonemes *'S'*, *'L'* and *'DH'* occur twice.

- **Assonance**: The phoneme *'ER0'* occurs twice.

- **Alliteration**: The phonemes *'S'* and *'L'* occur twice each at the start of the word.

However, this method can get us a lot of false positives. Common phrases like *'take away'* would give an assonance score on the *'EY1'* phoneme, which may not be intended. Therefore, we put a condition that we the occurrence counts for each sound device must be greater than 2. For example, consonance has a total occurrence score of six, assonance of two and alliteration of four. Therefore, the assonance would not be recognised.

Take another example: *"Mammals named Sam are clammy"*

- **Consonance**: *'M'* occurs five times while *'L'* occurs twice.
- **Assonance**: The phoneme *'AE1'* occurs thrice.
- **Alliteration**: None.

3.5 Form

We implement some detectors for the form and structure of the poem. This does not include analysis of words and phrases themselves as that will be done in the abstraction phase, as described in section BLAH [future abstraction section].

3.5.1 Structure and Repetition

The algorithms for detecting the features of structure described in section 2.1.2.4 are:

Number of stanzas
Count blank lines and add 1

Number of lines per stanza Count new line characters of each stanza

Number of distinct sentences The number of sentences returned when parsed using the CLiPS[22] *parsetree* function.
More lines than sentences indicates **enjambment**.

Number and location of repeated lines Find the list of non-unique lines. For each of those lines, find its line number locations in the poem.

The poem in Figure 3.1 has one stanza, four lines, no repeated lines and two distinct sentences, indicating enjambment.

The poem in Figure 3.2 has one stanza, five lines, no repeated lines and four distinct sentences, indicating enjambment.

3.5.2 Point of View

We can also determine the point of view of the speaker, i.e. whether it is in first or third person. If we find the word *'I'* anywhere in the poem and as long as it is outside speech marks, we say that the entire poem is in first person. Otherwise we default to third person. Figure 3.1 is written in third person but Figure 3.2 is in first person.

3.5.3 Tense

The tense of each line can be found by analysing the verb in that sentence. CLiPS pattern library provides a very easy method of doing this using their *'tenses'* function. It also gives us the *aspect*, e.g. perfect, progressive. However, tests found this to be less accurate so we will settle for just the tense.

We record the tense of each line as well as the overall tense of the poem. Each line in the cat poem in Figure 3.1 is in past tense giving a past tense overall. The limerick in Figure 3.2 is in present tense overall because all but one line, is in present tense.

3.6 Characters and Context

Here we describe the approach to the difficult challenge of determining the context of the poem in terms of its characters, their relationships, descriptions etc. as described in section 2.1.2.6. The aim is to build a representation of the characters much like a human reader would in their mind. This will then be compared in the abstraction phase to find a correlation between these representations and collections of poetry.

3.6.1 ConceptNet Relations

ConceptNet is a semantic network of common sense and general knowledge. Each node in the network is a natural language word or phrase called a *concept*. Edges in the network represent a relationship between two concepts. These relationships come in various types including, but not limited to:

MadeOf
What is it made of?
E.g. tree - MadeOf → wood

IsA
What kind of thing is it?
E.g. banana - IsA → fruit

AtLocation
Where would you find it?
E.g. priest - AtLocation → church

CapableOf
What can it do?
bird - CapableOf → sing

Desires
What does it want?
banker - Desires → his loan to be repay

HasA
What does it have in its possession?
old person - HasA → white hair

HasProperty
What properties does it have?
doctor - HasProperty → smart

PartOf

What is it part of?
player - PartOf → team

ReceivesAction

What can you do to it?
book - ReceivesAction → read

CreatedBy

How do you bring it into existence?
sound - CreatedBy → vibration

UsedFor

What do you use it for?
guitar - UsedFor → make music

MotivatedByGoal

Why would you do it?
learn - MotivatedByGoal → knowledge

Each of these relationships also has its inverse, e.g. NotIsA. We can use these relationships, along with a few of our own listed below, to build our desired representation.

Believes

What does the character perceive to be true?
Little Red Riding Hood - Believes → Wolf is her grandma

SendMessage

What did the character say or write?
Little Red Riding Hood - SendMessage → What a big mouth you have

ReceiveMessage

What did the character hear or read?
Little Red Riding Hood - ReceiveMessage → The better to eat you with!

TakesAction

What did the character do?
Wolf - TakesAction → Swallow Little Red Riding Hood

Named

What is the name of the character?
Gotham Vigilante - Named → Batman

MakesSound

What sound does the character make? Used for onomatopoeia detection 3.7.2
bird - MakesSound → tweet

We also have the corresponding inverses for each of these relations.

In keeping with the theme of this paper we cannot simply hard-code or randomise these relations when building a poem. We will instead analyse these relationships in existing poems and find correlations between them in the intepretation phase. For example, we may find that descriptive poems have a high number of *HasProperty* relations and very few characters.

To do this, we need to be able to extract ConceptNet-style relationships between concepts in a particular poem. If done correctly, the poem in Figure 3.1 will give us:

- cat - HasProperty → big
- cat - HasProperty → brown
- cat - Desires → eat a lot of mice
- cat - HasProperty → round
- cat - HasProperty → fat
- lot of mice - HasProperty → tasted so nice
- lot of mice tasted so nice - Causes → cat got all round and fat

The poem in Figure 3.2 will give us:

- limerick - TakesAction → packs
- laughs - ReceivesAction → packs
- laughs - HasProperty → anatomical

- space - HasProperty → quite economical
- laughs - AtLocation → space
- ones - HasProperty → good
- ones - NotHasProperty → clean
- ones - ReceivesAction → seen
- I - TakesAction → seen
- ones - ReceivesAction → seen
- ones - HasProperty → clean
- ones - NotHasProperty → comical

3.6.2 Semantic Labelling using Noah's ARK

It would be very difficult to determine these relationships from a syntactical parse alone. This is due to the complex nature of the English language, in particular verb usage. For example, the phrase *tasted so nice* is a description rather than an action because the word *tasted* in this case is being used as a linking verb, where it is usually an action verb.

These complexities are further compounded by the fact that we cannot rely on correct grammar in poems. We therefore need a *semantic* parse.

Noah's ARK is an informal research group run by Noah Smith at Carnegie Mellon University[4]. They provide online API access to two tools for linguistic structure analysis, *SEMAFOR*[14] and *TurboParser*[5]. Both of these tools can be used in conjunction to extract ConceptNet relations from natural language.

3.6.2.1 FrameNet Semantic Role Labelling using SEMAFOR

Semantic Role Labelling (SRL) is best described with an example. Take the sentence: *"The shopkeeper told the customer to have a nice day"* We wish to recognise the verb *'to tell'* as the coordinating word (called the **target**), *'the shopkeeper'* as the speaker, *'have a nice day'* as the message and *'the customer'* as the addressee. This output from SEMAFOR can be seen clearly in Figure 3.4, along with other potential labels.

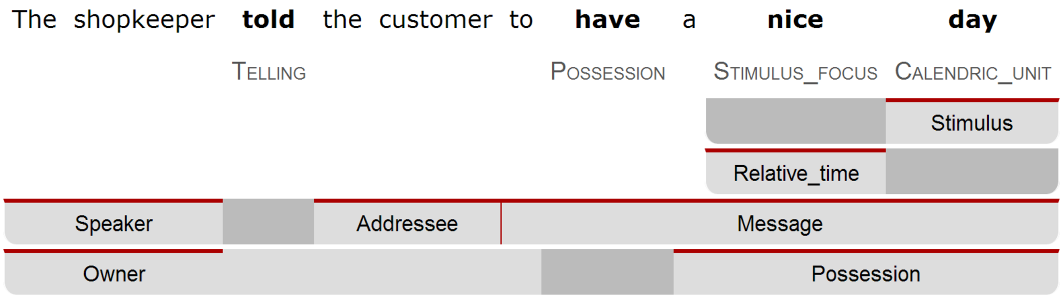


Figure 3.4: SEMAFOR frame-semantic parse for the shopkeeper example

This can be quite flexible as it is independent of the syntactic structure of the sentence and does require grammatical correctness. The SRL for the *"Yoda-speak"* equivalent will remain the same, as shown in Figure 3.5.

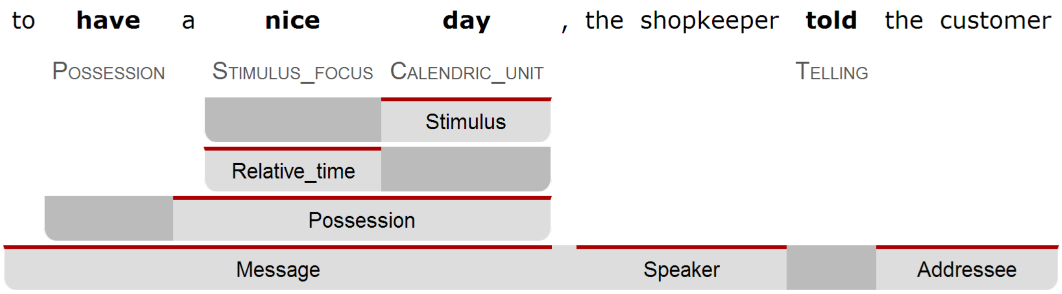


Figure 3.5: SEMAFOR frame-semantic parse for a grammatically incorrect sentence.

These labels are retrieved through the SEMAFOR tool, which was trained on FrameNet data to determine the frame-semantic structure of the text. FrameNet[6] is a lexical database of *semantic frames*, which describe the meaning of a word based on the words that typically participate with it, known as *frame elements*.

We can use these to derive the ConceptNet relations by looking for the occurrence of frames, and elements thereof, that correspond to the ConceptNet relation. The manually chosen list of frames and elements that translate directly into a particular ConceptNet relation is given in the Appendix, section A.5.

Each list may not be exhaustive for its corresponding ConceptNet relation and there are some relations that will not be picked up by this method.

3.6.2.2 Semantic Dependency Relations using TurboParser

The Stanford Dependencies[21] is another representation based around the relationships between words. All dependency relations are strictly binary and come in various types depending on the participants, called the *governor* and the *dependent*.

The semantic dependency tree for the previous shopkeeper example can be seen in Figure 3.6.

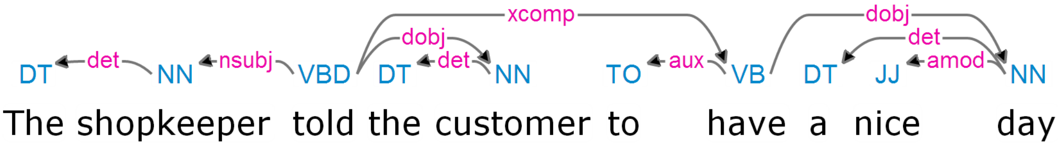


Figure 3.6: TurboParser Semantic Dependency parse for the shopkeeper example.

This representation fills the gaps left by the frame-semantic parse using the following heuristics:

governor - HasProperty → dependent
: If the dependency relation is *'amod'*, *'conj'* or *'poss'* or if the dependent is an adjective.

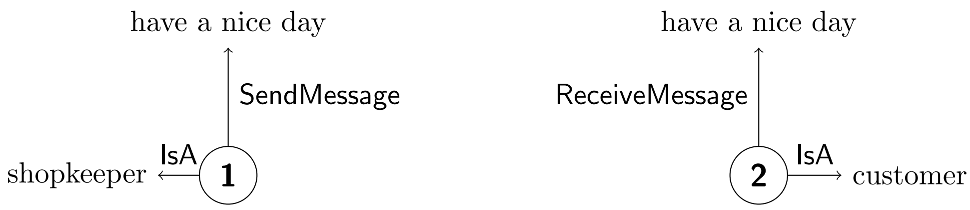
governor - atLocation → dependent
: If the dependency relation is *'agent'*, *'nsubj'* or *'prep'*, dependent is not a preposition, verb or 'WH' word and is a *location preposition* (in, at etc.).

governor - IsA → dependent
: If the dependency relation is *'agent'*, *'nsubj'* or *'prep'*, dependent is not a preposition, verb or 'WH' word and is not a preposition itself.

governer - ReceivesAction → dependent
: If the dependency relation is *'nusbjpass'* or *'dobj'* and the dependent is a verb.

governer - CapableOf → dependent
: If the dependency relation is *'xsubj'* or *'rcmod'*.

governer - TakesAction → dependent
: If the dependency is a verb whose lemma is not 'be' and no other relation has been found for this dependency.



(a) Hub for the 'shopkeeper' persona (b) Hub for 'customer' persona

Figure 3.7: Persona hubs diagram for shopkeeper example

Together, these tools provide fairly comprehensive coverage of the ConceptNet relations.

3.6.3 Extracting and Binding Relations to Characters

If we were to use the methods described above as they are, the derived ConceptNet relations would only be a set of abstracted FrameNet frames and matched dependencies. This alone does not give us much more information than if we were to use the frame-semantic parse on its own. The true usefulness of this approach only arises if the relations can be **bound** to persona in the poem.

For the shopkeeper example, we would recognise *'the shopkeeper'* and *'the customer'* as **persona** with *SendMessage* and *ReceiveMessage* relations bound to each of them with respect to the *'have a nice day'* message.

To accentuate the persona-centred structure, we can represent the desired output as a set of 'hubs', with each persona at the centre of the hub. Figures 3.7, 3.8, 3.9 show this for the shopkeeper, cat poem and limerick examples respectively.

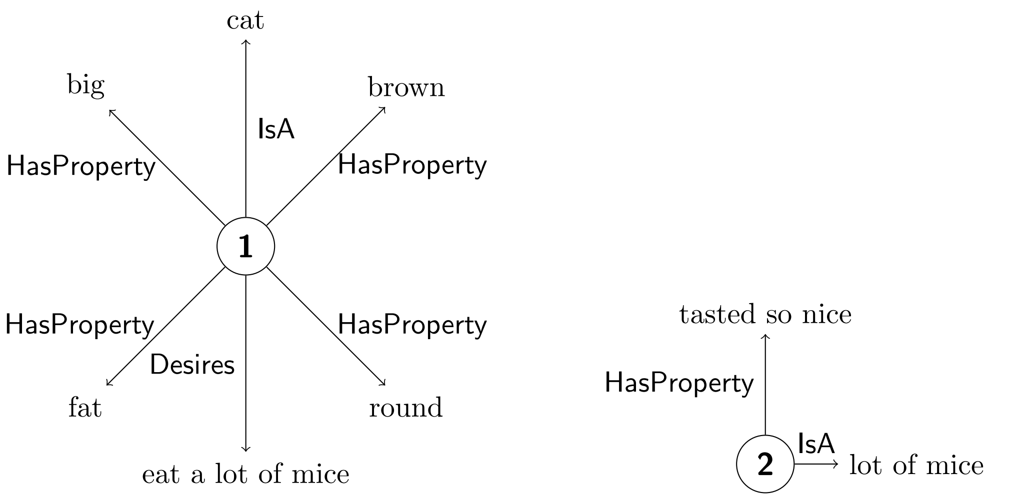
This will also improve our knowledge for the generation phase in that we will know the *number of characters* typical for a collection of poems, the *number and type of relations* associated with *each* character, as well as allow us to find commonalities in the types of characters themselves.

To reach the desired representation, we execute Algorithm BLAH:

1 for each sentence in the poem:

2 get the dependency relations and frame-semantic parse

3 collapse loose leaves of the dependency relations



(a) Hub for the 'cat' persona (b) Hub for the 'lot of mice' persona

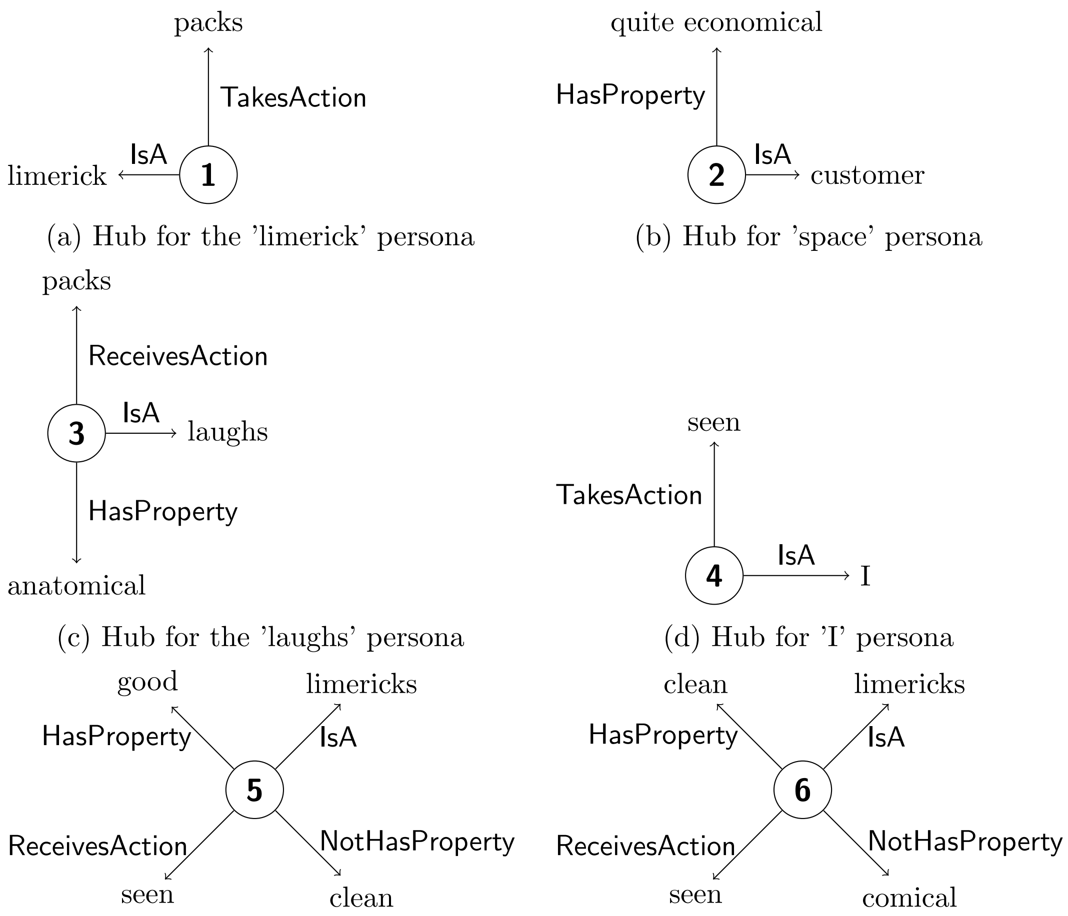
Figure 3.8: Persona hubs diagram for cat poem example

4 find and create possible character objects
5 find candidate ConceptNet relations from the frame-semantic parse
6 for each character object:
7 get all associated dependency relations
8 for each associated dependency relation:
9 if the dependent is involved in a candidate relation from the frame-semantic parse:
10 bind the relation to the current character object and continue
11 else:
12 use the heuristics for dependency relations to find possible ConceptNet relations
13 bind any that are found to the current character object and continue

We will describe each step in detail.

3.6.3.1 Obtaining the Semantic Dependency Relations and Frame-Semantic Parse from Noah's ARK

Noah's ARK provides parse data from both SEMAFOR and TurboParser in a JSON format that can be accessed by sending a request to their demo API. We leave the frame-semantic parse from SEMAFOR in this format because we only need to access



(a) Hub for the ’limerick’ persona

(b) Hub for ’space’ persona

(c) Hub for the ’laughs’ persona

(d) Hub for ’I’ persona

(e) Hub for the persona represented by the first occurrence of ’ones’

(f) Hub for the persona represented by the second occurrence of ’ones’

Figure 3.9: Persona hubs diagram for limerick example

this data once when finding the candidate relations on line 5 of Algorithm BLAH.

TurboParser returns the Semantic Dependency Parse in CoNLL data format, whose structure can be seen in Table 3.1.

We parse this into a dependencies dictionary format shown in Table 3.2 for easy access and processing downstream.

TurboParser and SEMAFOR are resource heavy (SEMAFOR requires minimum 8GB RAM), so it makes sense to run it on a server. Since the output from Noah’s ARK is exactly what we want, the request is fairly quick (typically 1.24 seconds for simple sentences and 1.61 seconds for complex ones, tested using the specifications in Appendix A.4) and speed is not a priority for this project, there is no need to run a copy on our

Field Number	Field Name	Description
1	ID	Token counter, starting at 1 for each new sentence.
2	FORM	Word text form or punctuation symbol
3	LEMMA	Lemma or stem of FORM
4	CPOSTAG	Coarse-grained part-of-speech tag
5	POSTAG	Fine-grained part-of-speech tag
6	FEATS	Unordered set of syntactic and/or morphological features
7	HEAD	ID of the parent of the current token ('0' if root)
8	DEPREL	Dependency relation to the HEAD
9	PHEAD	Projective head of current token
10	PDEPREL	Dependency relation to the PHEAD

Table 3.1: The CoNLL data format output by TurboParser.

ID	FORM	CPOSTAG	POSTAG	HEAD	DEPREL
1	There	EX	EX	3	expl
2	once	RB	RB	3	advmod
3	was	VB	VBD	0	null
4	a	DT	DT	7	det
5	big	JJ	JJ	7	amod
6	brown	JJ	JJ	7	amod
7	cat	NN	NN	3	nsubj
8	who	WP	WP	9	nsubj
9	liked.	VB	VBD	7	rcmod
10	to	TO	TO	11	aux
11	eat	VB	VB	9	xcomp
12	a	DT	DT	13	det
13	lot	NN	NN	11	dobj
14	of	IN	IN	13	prep
15	mice	NN	NNS	14	pobj
16	.	.	.	3	punct

Table 3.2: The dependencies dictionary data structure for the first sentence of the cat poem.

own server.

3.6.3.2 Collapsing Loose Leaves of the Semantic Dependency Relations

In a lot of cases, the character or relation that we look for is a phrase, not just a word. For example in Figure 3.10 the character should be *'a lot of mice'* rather than just the noun *'mice'*. In fact if we skipped this step, we would get two characters - *'lot'* and *'mice'* - which is obviously wrong.

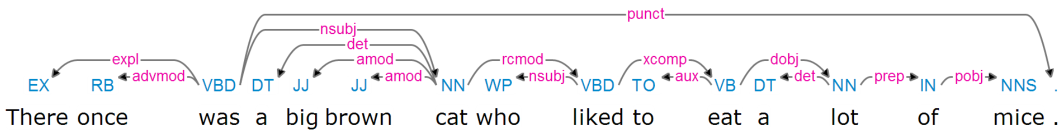


Figure 3.10: TurboParser Semantic Dependency parse for the first sentence of the cat poem.

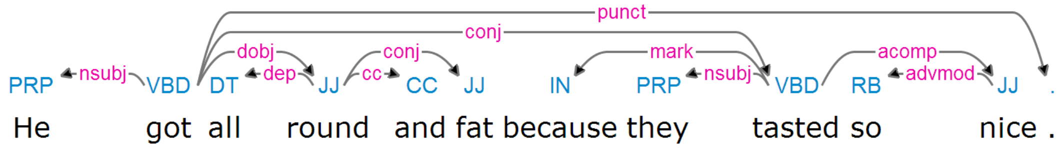


Figure 3.11: TurboParser Semantic Dependency parse for the first sentence of the cat poem.

Another example is the phrase *tasted so nice* as in Figure 3.11. If we did not collapse the tree, we would have that *'they'* took the action of *'tasted'* and can be described as *'nice'*, both of which are wrong again.

To solve this, we merge leaves of the tree together if the dependency is *'collapsable'* as per a set of conditions listed in the Appendix, section A.6.

By default, the parent of the dependency keeps all its attributes unchanged except for merging its form with that of the leaf, e.g. *'of'* and *'mice'* becomes *'of mice'*. However, there are some cases where we wish to retain the part-of-speech (POS) tag of the leaf and overwrite the parent's POS tag:

- If the leaf is an adjective, the parent is a verb and the dependency relation is *'dep'* then we retain the adjective POS tag. These conditions usually imply a *linking verb* generally used to describe a property, not an action. E.g. 'tasted so nice' is an adjectival phrase despite the use of a verb.

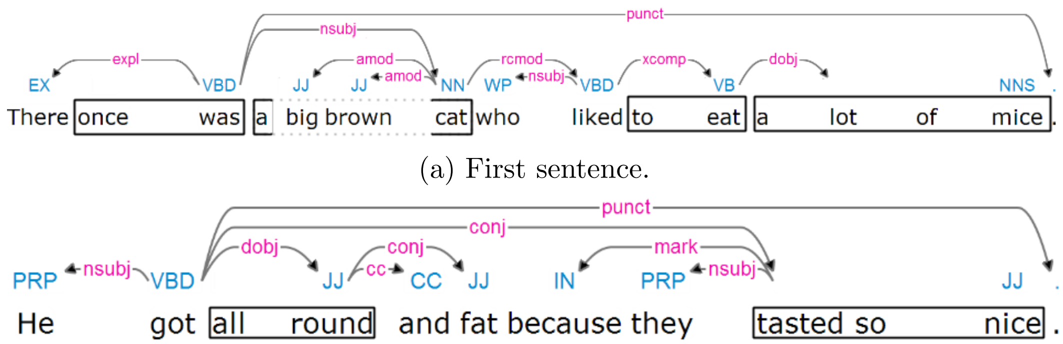
- Collapsing an *'acomp'* dependency relation should also retain the child POS tag because it too is evidence of a linking verb.

- The *'pobj'* and *'prep'* dependency relations are evidence of prepositions, which often link words together that we do not want to keep separate. For example, *of mice* should retain the 'NNS' POS tag of *mice* rather than keep the 'PRP' POS tag of *of*.

Then we follow Algorithm BLAH:
1 Get a list of all the leaves in the graph.
2 For each leaf in the reverse of this list:
3 If the dependency relation of this leaf (i.e. from its parent to it) is collapsable:
4 Get the parent
5 Merge the form of the parent and the leaf
6 If necessary, the parent retains the part-of-speech tag of the leaf
7 The leaf is destroyed and the parent becomes the leaf.
8 Loop back to line 3.

Figure 3.12 shows a dry run of this algorithm on the first line of the cat poem as shown in Figure 3.10.

The final diagrams of the collapsed dependencies for the full cat poem example can be seen in 3.13



(a) First sentence.

(b) Second sentence.

Figure 3.13: Final collapsed dependency diagrams for full cat poem example.

3.6.3.3 Finding and Creating Characters

A naive method for finding persona would be to simply extract nouns and pronouns. This would actually be sufficient for a basic level of analysis, but we can do better. Notice that in the second half of the cat poem, the pronoun *'he'* is used to refer to the 'cat' persona and *'they'* to the 'a lot of mice' persona.

In Computational Linguistics, the problem of matching pronouns, or more generally *anaphora*, with the *antecedent* (or *postcedent*) that it refers to is called ***Anaphora Resolution***. It is an ongoing research area and I will present a new potential solution in section 3.6.4.

Part of the solution is gathering some basic semantic information about the character. We would like to determine whether the phrase representing the character:

1. is plural or singular.
2. is male, female, neutral or unknown.
3. is a living object, an inanimate physical object or a non-object.

Determining the first point is easiest:

- If it is a noun and the POS tag ends in an 'S', it is plural (see *'mice'* tag in 3.10).
- If it is a pronoun, check for membership in a manually built list of plural pronouns (e.g. 'they', 'them').

The pronoun case for the second point is similar; we check for membership in the manually built list of male pronouns (e.g. 'he', 'him') and the separate list of female pronouns (e.g. 'she', 'her').

The noun case is trickier. First we need to find the *synset* of the noun. Synsets are sets of cognitive synonyms, e.g. *car* and *automobile* are in the same synset, but not *cable car*. Once we have this, we can use the *hypernym* relationships between synsets.

A hypernym of a synset is a **type-of** relation. For example, the synset *mammal* is a hypernym of *cat* because cats are mammals. Similarly, *animal* is also a hypernym of *cat*, as well as being a hypernym of *mammal*. The full hypernym tree for *cat*, according to lexical database WordNet, can be seen in Figure 3.14.

The *direct hypernym* of a synset is the synset directly above it in the hypernym tree; *feline* in the case of the cat. The *inherited hypernym* of a word refers to any word that appears in its hypernym tree. WordNet[38], a lexical database of the cognitive relations between words provides all of these resources.

Now, finally, we can check for whether a noun is male or female by looking for the existence of particular synsets that imply a gender. For example, the *'female'* synset would be an inherited hypernym of the synset *'cow'*. Therefore we know that cows are female. Similarly, *'maharaja'* has the hypernym *'prince'*, which we know to be male.

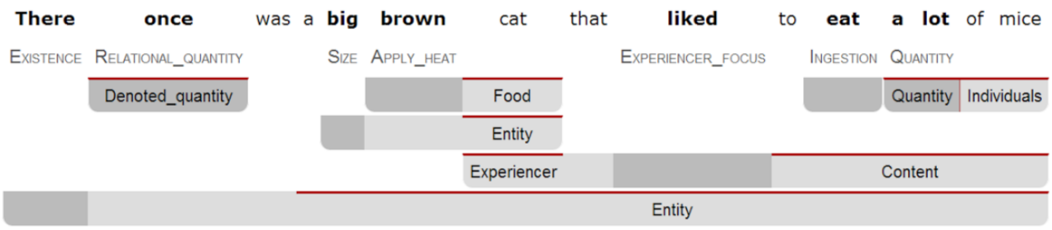
We can extend this practice to the final point of determining the animation of the character by checking if the *'living thing'* and *'physical object'* synsets are inherited hypernyms of the synset of the noun concerned.

3.6.3.4 Extracting Candidate Relations from the Frame-Semantic Parse

Using the map between FrameNet frames and ConceptNet relations mentioned in section 3.6.2.1, we can carry out Algorithm BLAH.

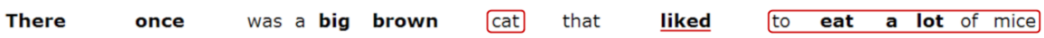
1 for each frame found in the frame-semantic parse:
2 look up the target of the frame in the map
3 if it can help build a ConceptNet relation:
4 retrieve the frame elements
5 replace the frame elements with the corresponding text from the poem
6 if we cannot find all the elements we are looking for, we leave it blank
7 add mapping from the text of the target to the newly built relation so we can find it later

Figure 3.15 show a dry run of this algorithm using the first half of the cat poem in Figure 3.1.

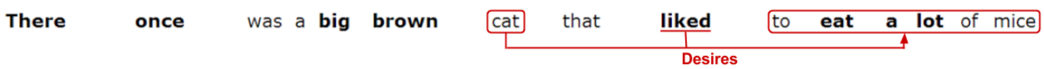


(a) Get all the frames found by the frame-semantic parse

(b) The only one that we can convert to a relation is the Experiencer_Focus frame



(c) Retrieve the frame elements with an anchor on the target text, *'liked'*



(d) Create the corresponding relation and map it to the target

Figure 3.15: Execution of Algorithm BLAH on the first half of the cat poem example.

No frames could be converted into ConceptNet relations for the second half of the cat poem example.

3.6.3.5 Obtaining the Associated Dependency Relations for each Persona

This step breaks up the semantic dependency tree and flattens it into the character-centric hubs like the ones in Figure 3.8. We start from Figure 3.12g.

First, we want to identify the persona in the dependency tree. All of the relations going out from it are naturally related dependencies of the persona, so they get added to the hub.

The single relation coming into this character is also a related dependency. We reverse the direction of the branch and add it to the hub. The hub so far is more like a spider diagram, as shown in Figure 3.16.

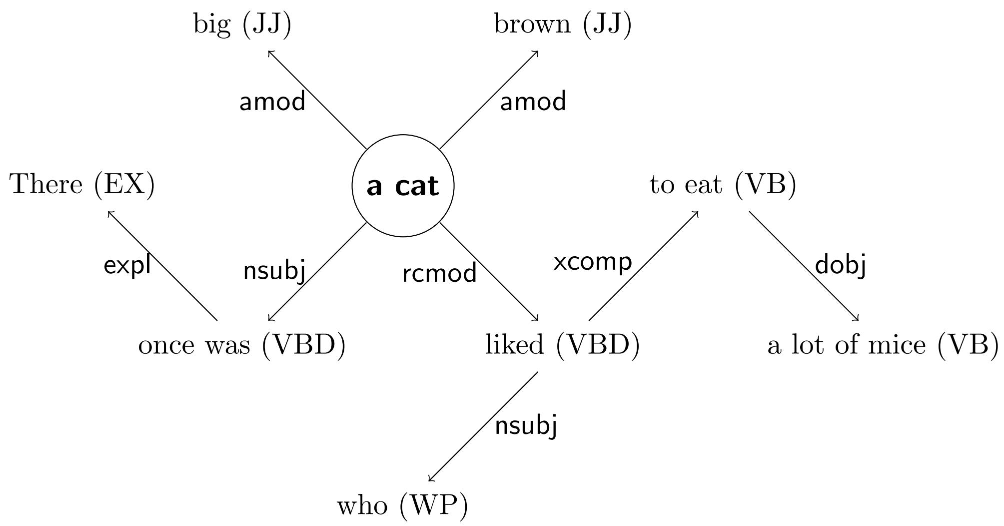
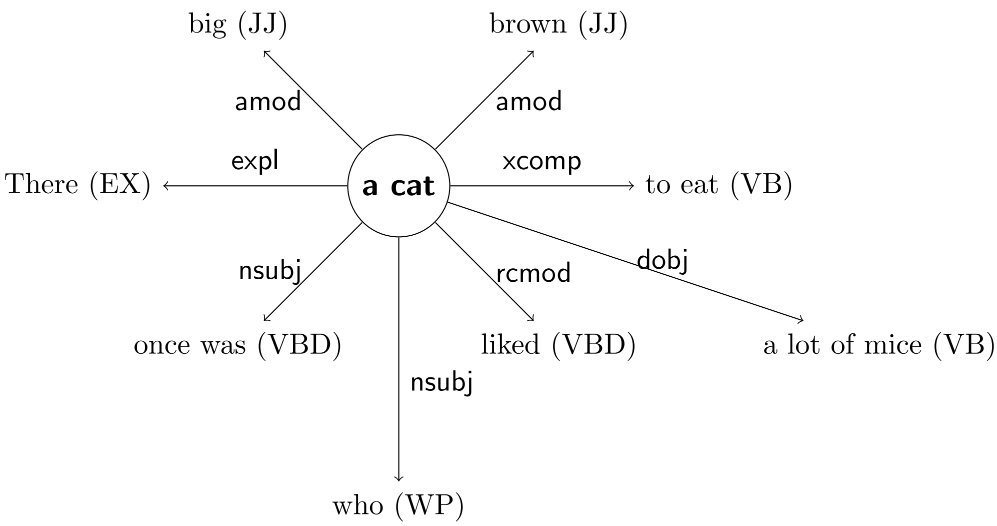


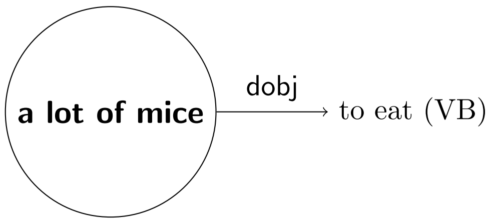
Figure 3.16: Initial dependency hub for the 'cat' persona.

Now we have a tree with the character as the root. The next step is to flatten it so that everything is related to the character. We do this by recursively converting all the grandchildren of the character root node to a direct child node until there are no more grandchildren. This is shown in Figure 3.17.

We repeat this process for each persona starting from the leftmost character in the sentence. The persona hubs for the second half of the poem is shown in Figure 3.18.

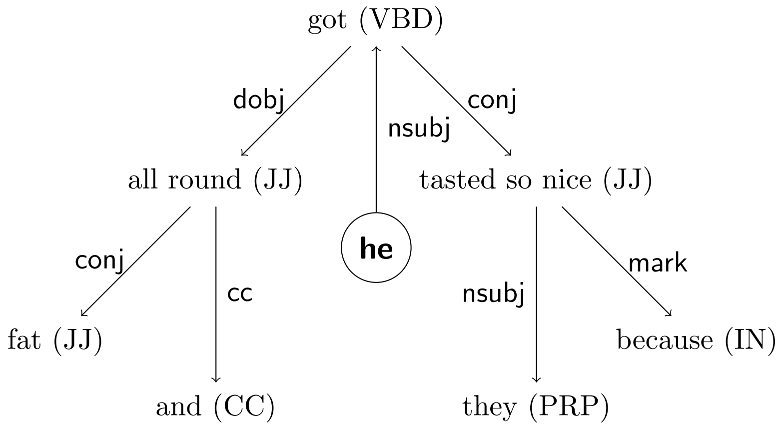


(a) Persona: 'cat'

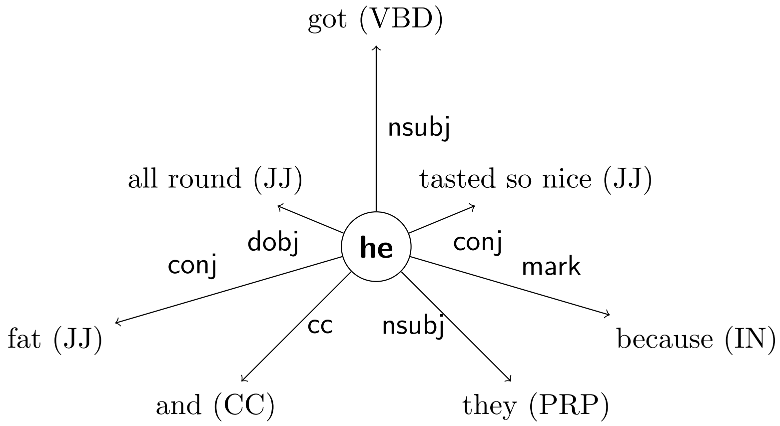


(b) Persona: 'a lot of mice'

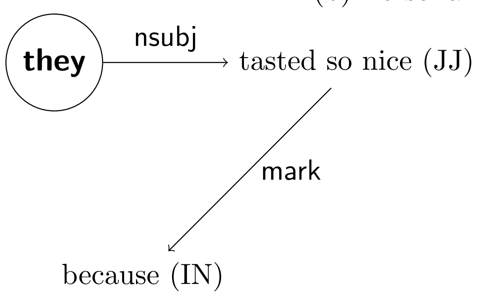
Figure 3.17: All persona hubs for the first sentence of the cat example.



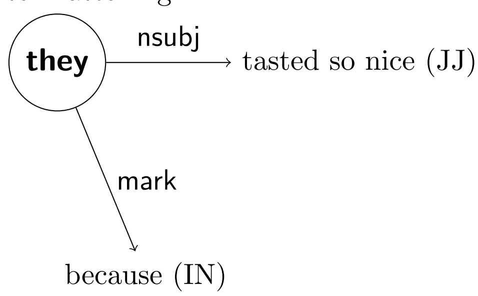
(a) Persona: 'he' before flattening.



(b) Persona: 'he' after flattening.



(c) Persona: 'they' before flattening



(d) Persona: 'they' after flattening

Figure 3.18: All persona hubs for the second sentence of the cat example.

Notice that the same dependency *'mark'* dependency with *because (IN)* is associated to both *'he'* and *'they'*. The *tasted so nice (JJ)* node is also involved with both of those nodes, although the type of dependency differs.

In most cases, we do not want either of these duplicated dependencies as it can lead to redundant relations. We will deal with this in the next step where we convert these dependencies into relations.

3.6.3.6 Binding ConceptNet Relations to Characters

Now that we have our hub data structure for all persona, we now need to look at each branch of each hub and decide if it can be converted into a ConceptNet relation.

First, we check if the text of the child maps to the target of one of the candidate relations we extracted from the frame-semantic parse in section 3.6.3.4. If it does, then we accept that and move on to the next child without checking the dependency relation since it is likely to be the less accurate.

Sometimes the dependant of the relation will be blank because we could not find the right frame element as explained earlier. In this case, we just assume that this is a relation to the next persona in the list. This is a fair assumption given the limited number of persona per sentence, the probable forward reference structure of sentences and the fact that most of the relations we look to build are between persona.

If there is no candidate relation for this child, we then look at the type of the dependency between it and the character. We use the heuristics described in section 3.6.2.2 to convert it into a ConceptNet relation.

If we cannot find a relation through either of the above methods then it is likely there isn't one, so we remove it from the hub and move on to the next child.

In all of those steps, we need to look out for negative adverbs such as 'not', 'seldom', 'rarely' etc. which we use to negate the ConceptNet relation found. We also need to look out for antonyms in the target word in the frame-semantic parse. For example, the *Experiencer-focus* frame helps us find *Desires* and *NotDesires* ConceptNet relations depending on whether the target word is synonymous with 'love' or with 'hate'.

To avoid the aforementioned duplication problem, we prune these hubs by working backwards through the persona (i.e. starting from the rightmost in the sentence) and removing any relations that occur in earlier persona. So in fact the *'lot of mice'* persona has no relations (before anaphora resolution, see Section 3.6.4).

The duplication is mostly due to the reversing of direction on incoming dependencies (heads). This pruning method works because the head of the last persona will be lower down in the dependency tree than the head of any other persona. Duplicated relations are removed such that it is solely associated to the one it was closest to in the original tree.

We can see this entire process being applied step-by-step to the 'cat' persona in our ongoing example in Figure ??. The final hubs for all persona can be seen in Figure ??.

3.6.4 Anaphora Resolution

We can now clearly see the anaphora problem as described in section 3.6.3.3. We must match up the *'a cat'* character and *'he'* character, and the *'they'* character with the *'a lot of mice'* character.

In this particular example, we can simply match the plurals and singulars together. However, we may not always be so lucky. We may also run into a situation where the cat could be referred to as *'the feline'* instead of *'he'*.

Section 2.4.5 gives an overview of the state of the art solutions. Our solution is also pretty darn simple at the moment, so I am going to wait to see if I can do some of the more complicated things and then write this section up.

3.6.5 Peeks at Future Development of Pragmatic Language Analysis

If I get time...

3.7 Symbolism and Imagery

This particular poetic technique can be very subtle and difficult to detect. Of the ones listed in section 2.1.2.5, we are able to recognise the use of three, similes; onomatopoeia and personification.

3.7.1 Personification

We used the animation state of characters for anaphora resolution. We can take this idea further to detect the use of personification in poetry.

The following relations are symptomatic of a sentient being:

- 'Named' and 'NotNamed'
- 'Desires' and 'NotDesires'
- 'Believes' and 'NotBelieves'
- 'SendMessage' and 'NotSendMessage'
- 'ReceiveMessage' and 'NotReceiveMessage'

Therefore, if a character that is marked as an inanimate physical object has any of these relations, it is likely that the author of the poem was using personification to give the object human-like behaviour.

We can also take advantage of the *Semantic Type* attribute of FrameNet elements. FrameNet marks frame elements with a semantic precondition if they are to be of a certain type, for example only *'Sentient'* semantic types can be the *'Abuser'* and the *'Victim'* in the *'Abusing'* frame. By looking up these semantic types in FrameNet as we are building the ConceptNet relations, we are able to flag characters for personification.

3.7.2 Onomatopoeia

Some onomatopoeia are recognised in traditional dictionaries, while some are not. There are also many variations of the same sound, e.g. 'Aah' versus 'Aaah'. New onomatopoeia can be created at any point depending on the sound the writer wants to portray. This range of variation and lack of consistency makes it difficult to detect all occurrences of onomatopoeia.

WrittenSound.com is an online dictionary devoted to onomatopoeia. We use this to recognise the most common ones. A word in the poem is an onomatopoeia if the closest matching word in the dictionary, using *difflib*, has a similarity score of greater than 0.9

We check for onomatopoeia while building ConceptNet relations. If we detect onomatopoeia, we add the MakesSound relation to the character with respect to the onomatopoeia. For example *"the window rattled"* would add the relation *MakesSound* → *rattled* to the *'window'* character object.

Furthermore, WrittenSound provides mappings between the onomatopoeia and the type of sound they represent. For example, *'ding'* is mapped to *'hard hit'* and *'metal'*. We add extra bonus relations by making deductions on this. For example, if the character makes a *'ding'*, sound, we can deduce and add the relations *ReceivesAction* → *hard hit* and *MadeOf* → *metal*.

These deductions are chosen manually. The full list is in the Appendix section A.7

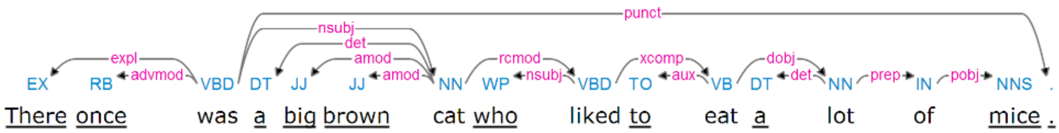
3.7.3 Simile

Similes are relatively easy to detect because they often use the phrases 'like a' or 'as a' and 'than'. They also follow certain syntactical patterns, as in Figure BLAH.

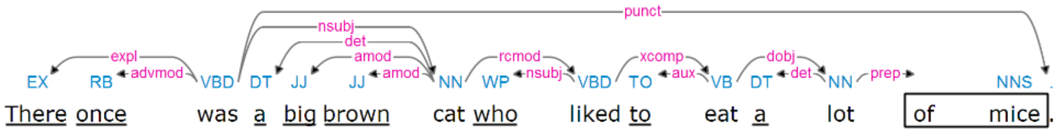
Figure: in that it is usually a noun followed by a verb or adjective, then one of the aforementioned phrases, followed by a noun or an adjective.

A particular symptom of simile use is that the aforementioned phrases will be involved in a prepositional noun phrase, or 'PNP Chunk' to use the parser terminology.

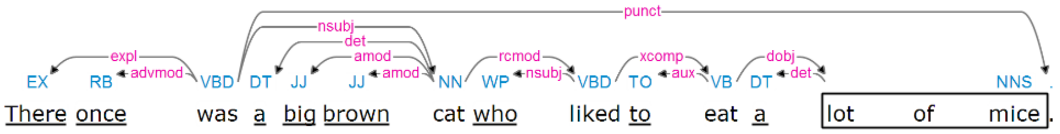
We could take this further by using ConceptNet btw. As fast as a cheetah + cheetah HasProperty fast -¿ simile.



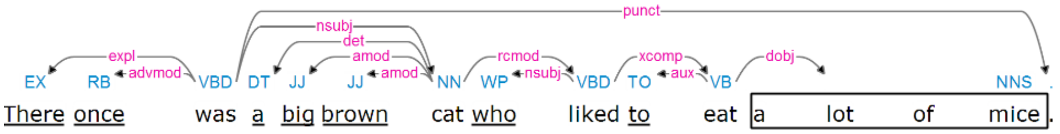
(a) Identify all of the leaves



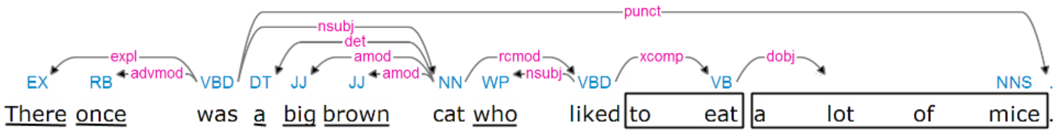
(b) Start from the rightmost leaf, *'mice'* (ignore punctuation). Dependency *'pobj'* is collapsable and we inherit the child POS tag.



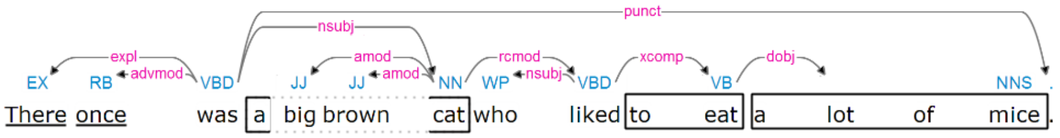
(c) Because we collapsed a leaf in the previous step and it remained a leaf, we see if we can collapse the same leaf further. Dependency *'prep'* is collapsable and we inherit the child POS tag once again.



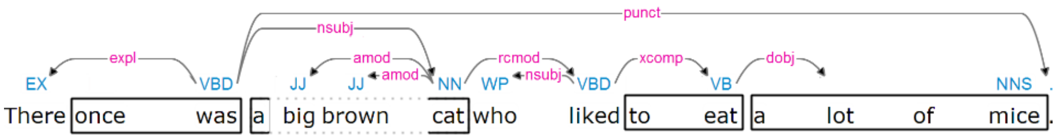
(d) The leaf we collapsed is now a parent so we move left to the next leaf *'a'*. Dependency *'det'* is collapsable and we retain the parent POS tag this time.



(e) We have a leaf again, but we cannot collapse any further. We move on to the next leaf, *'to'*. Dependency *'aux'* is collapsable and we retain the parent POS tag.



(f) Move on to next leaves. Dependencies *'nsubj'* and *'amod'* are not collapsable, but *'det'* is so we collapse leaf *'a'* with its parent *'cat'*.

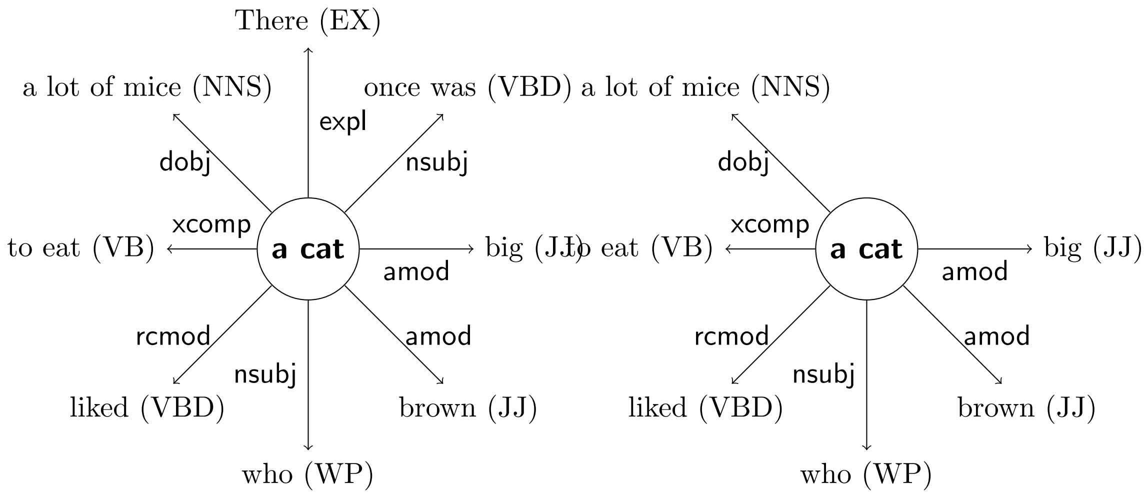


(g) Dependency *'advmod'* is collapsable and *'expl'* is not. Collapse accordingly and finish.

Figure 3.12: Execution of Algorithm BLAH on the first half of the cat poem example.

cat
└ feline
└ carnivore
└ placental
└ mammal
└ vertebrate
└ chordate
└ animal
└ organism
└ living thing
└ whole
└ object
└ physical entity
└ entity

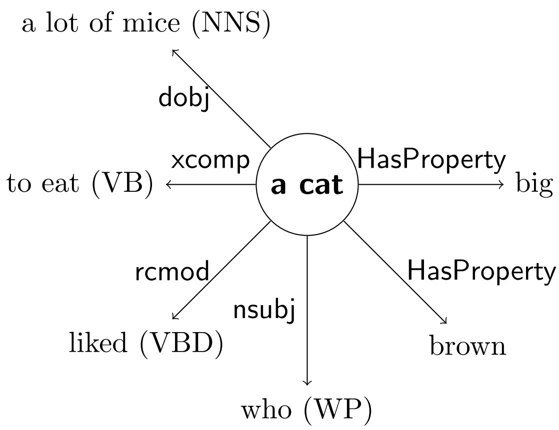
Figure 3.14: Full inherited hypernym tree for 'cat' synset according to WordNet



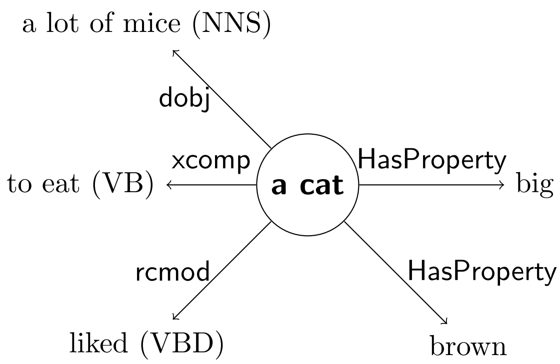
(a) Dependency hub for the 'cat' character, rearranged such that the children appear in the order they do in the original sentence, starting from *'There'* in the 12 o'clock position in a clockwise direction.

(b) Dependency *expl* does not map to any relation and the child *'There (EX)'* does not map to any relation via the frame-semantic parse. It is removed. Similarly remove the *'once was (VB)'* child because it is a verb but the lemma includes the verb 'be'.

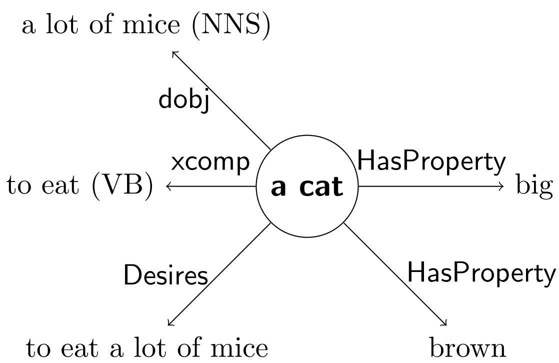
Figure 3.19: Full binding process for the 'cat' character



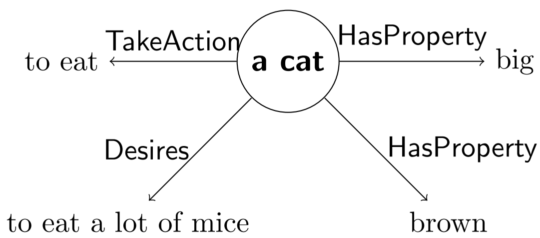
(c) Dependency *'amod'* maps directly to a HasProperty relation for both the *'big (JJ)'* and *'brown (JJ)'* children.



(d) Remove the *'who (WP)'* child because it has a POS tag starting with W so the *'nsubj'* relation does not matter.

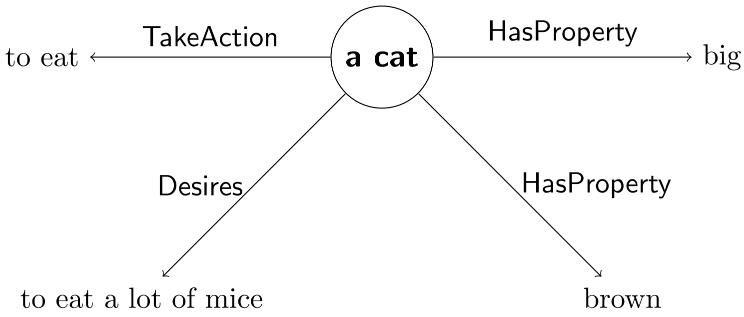


(e) In this case, *'liked (VBD)'* is a target mapped to a relation found from the frame-semantic parse. It gets replaced by this relation with no further consideration.

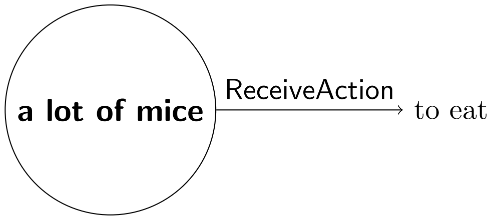


(f) The final *'a lot of mice (NNS)'* child is not a verb so the *'dobj'* relation has no effect.

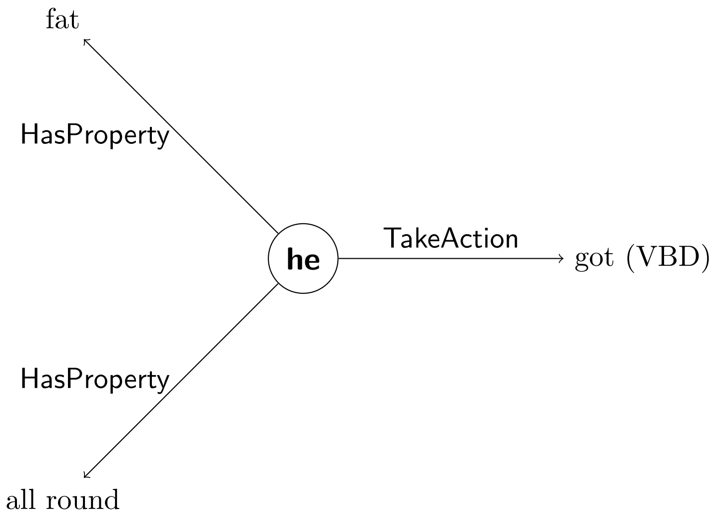
Figure 3.19: Full binding process for the 'cat' character



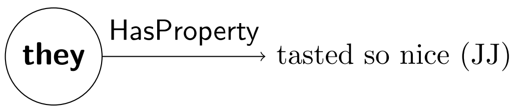
(a) Bindings for 'cat' persona.



(b) Bindings for 'lot of mice' persona.



(c) Bindings for 'he' persona. Notice *'tasted so nice (JJ)'* was removed from here because of duplication with *'they'* persona below. We get one anomaly with *'got'* because we were unable to distinguish between definitions of 'became' and 'retrieved'.



(d) Bindings for 'they' persona.

Figure 3.20: Complete bindings for all characters

Chapter 4

Analysis Interpretation

The second phase of implementation involves interpreting the first phase analysis results. The goal is that these interpretations can be used to template the structure and outline the content for certain collections of poems, ultimately guiding the generation of novel and authentic poetry in the final phase.

4.1 Balancing Priorities and Trade-offs

In deciding how to approach this phase, we need to make predictions about the usage of the final interpretation and assumptions about the quality, or lack thereof, of the corpus and analysis. Doing this will outline the priorities and trade-offs in implementation that will most benefit the generation phase.

4.1.1 The Corpus

The corpus is a large collection of poems. We may assume that each poem is already marked with the type of the collection - limerick, haiku, riddle etc. - much like data in a supervised machine learning problem. However, the poems should otherwise be *unseen* before analysis; we know nothing else about them.

Poems are most easily obtained from anthologies. This works very well with our assumptions as the data is already labelled with the type of poem, but we know nothing else about the content.

Another way to gather poems is by looking for those freely available on the Internet. However, the corpus obtained from an Internet search will likely be of lower quality than directly from a published anthology. For example, we may get the limerick in Figure 4.1, which is clearly very awkward. We can immediately see that it does not follow the typical *'AABBA'* rhyme scheme, and even more so that it doesn't have a fifth line!

A limerick fan from Australia
regarded his work as a failure:
his verses were fine
until the fourth line
?

Figure 4.1: A very awkward limerick

Nevertheless, our blindness to the content of the corpus means the system must be robust to such outliers. Therefore, we *will* get anomalous results and need to be able to cope with them.

4.1.2 Subcategories of Collections

The first two stages aim to be an exhaustive analysis of any type of poem. The first stage achieves this by detecting as many features in a single poem as possible. The second stage does this not only by finding highlights among features of homogeneous poems, but correlations between them as well.

For example, limericks that start with *'There was a...'* may have a very distinctive rhythm pattern for the first line that is otherwise not strictly followed. Therefore, when generating new poetry in the next phase that starts with *'There was a...'*, we want to make sure we abide by this correlation.

There will be many correlations with more than two variables that may be more difficult to spot, but the complexity of finding all of them is exponential. We need to find a balance between usage of this data and the complexity in obtaining it.

4.1.3 Novel and Creative Generation

There's another catch to the correlation problem - red herrings. Some features can be taken as law; 5 lines in a limerick, 5-7-5 syllable pattern in Haiku, iambic pentameter in Shakespearean Sonnets. However, some may just be coincidences or the result of a very biased corpus.

Furthermore, we want to be able to create poems that are *novel* and *creative*. So while we need to be guided by prior art on structure and content to create authentic pieces of literary art, we need to balance that with freedom of creativity.

Perhaps we can use this stage to give us an idea of when and how we are allowed to diverge.

4.2 A Stochastic Approach

The most suitable implementation that meets the priorities listed above is:

- Analyse each poem individually as planned.
- Store collections of poems by type (limerick, haiku etc.). This avoids extra filtering and repeated memory persistence downstream.
- Put the analysis results of each poem in a particular collection together, keeping each feature independent. For example, we put all of the rhyme schemes for all limericks in one big list, saving it separately from the list of stanza lengths. I will refer to this step as *'aggregation'* from now on.
- For each feature, create a probability distribution of the results that represents the likelihood of any particular value occurring for each feature. If 95 out of 100 limericks have an $AABBA$ rhyme scheme, then we consider the probability of an $AABBA$ rhyme scheme in a limerick to be 95%.

The aggregation step can be bespoke to each feature. For example, we would aggregate the tenses for each line of each poem differently to the way we would aggregate the overall tense for each poem. This enables us to handle outliers on a case-by-case basis since they are defined differently for every feature.

We can also decide how to interpret the data on a case-by-case basis, helping us handle the varying error bands for each feature. For example, detecting the rhyme scheme is more error-prone than detecting the number of syllables so we may round a 95% result for rhyme scheme up to 100%, but take it literally for syllable pattern.

This implementation also gives an effective way of finding correlations. We take a value for one of the features as a *given*, filter the store of analysed poems by this value and aggregate in the same way. We can take any number of given values and of any permutation since each feature is aggregated and interpreted in isolation.

As a bonus, this implementation quite efficient both in terms of time and space (see Section 4.5). By storing the original poem analysis results, we can apply given values and find correlations *lazily*, i.e. only when needed.

4.3 Algorithms

There are four main algorithms that are used for aggregation. All features can be aggregated using some variant of these algorithms, with slight feature-specific amendments so to have the flexibility and precision mentioned in the previous sections.

4.3.1 List Comprehension

The simplest of the four algorithms utilises Python's built in functional programming feature; list comprehensions. In a single line of code we put all of the results for a particular feature from all poems in the collection into one big list. Algorithm BLAH shows the code for aggregating the number of stanzas.

1. *num_stanzas = [poem.n_stanzas for poem in poems]*
2. *return Count(num_stanzas)*

The second line gets the frequency distribution of this list, which can easily be turned into a probability distribution as required.

4.3.2 Line by Line

Sometimes we can gain more insight by interpreting patterns in a feature for **each line** rather than the poem altogether. In these cases, the analysis result is usually saved in the form of a tuple, one entry for each line.

Take stress patterns for example. The analysis of a single poem will store the result as `(x`$_1$`, x`$_2$`, ..., x`$_n$`)` where `x`$_i$ is the stress pattern for line `i` up to the total number of lines in that poem, `n`.

Suppose we have three poems in the corpus altogether. Then we have three tuples to aggregate: `(x`$_1$`, x`$_2$`, ..., x`$_n$`)`, `(y`$_1$`, y`$_2$`, ..., y`$_n$`)` and `(z`$_1$`, z`$_2$`, ..., z`$_n$`)`.

If we are looking generalise the results on a line by line basis, we want to **transpose** the data to group each line together and put them in a list with the format `[(x`$_1$`, y`$_1$`, z`$_1$`), (x`$_2$`, y`$_2$`, z`$_2$`), ... ,(x`$_n$`, y`$_n$`, z`$_n$`)]`.

Once this is done, we can apply the list comprehension algorithm described in Section 4.3.1 to each tuple in the list. Algorithm BLAH outlines this full process with the stress pattern example.

1. *stress_patterns = [poem.stress_patterns for poem in poems]*
2. *stress_patterns_all_lines = transpose(stress_patterns)*
3. *return [Count(stress_pattern_single_line) for stress_pattern_single_line in stress_patterns_all_lines]*

4.3.3 Pick the Most Popular

The analysis of a single poem often has some ambiguities. For example, the rhyme scheme for a collection can never be decided by looking at just one poem because words can be pronounced in more than one way. The limerick in Figure 4.2 could either be *AABBA* or *AABBC* because *invited* can be pronounced as *('IH', 'N', 'V', 'AY', 'T',* ***'AH'****, 'D')* or *('IH0', 'N', 'V', 'AY', 'T',* ***'IH'****, 'D')* according to the CMU pronunciation dictionary.

There was a young man so benighted
He never knew when he was slighted;
He would go to a party
And eat just as hearty,
As if he'd been really invited.

Figure 4.2: A limerick with more than one possible rhyme scheme

The analysis phase doesn't make a claim on which rhyme scheme is 'correct' and instead stores all possibilities in a list `[x`$_1$`, x`$_2$`, ..., x`$_n$`]`, where `x`$_i$ is the i^{th} rhyme scheme possibility up to `n` possibilities for the particular poem being analysed.

Let's assume again that we only have three poems in the corpus. Once they are all analysed, we get three lists [x_1, x_2, ..., x_n], [y_1, y_2, ..., y_n] and [z_1, z_2, ..., z_n] of possible rhyme schemes for all three poems.

We illustrate what we want to happen by taking four cases:

- None of the possibilities across all poems are the same. Then all are equally probable and that gives us our probability distribution.
- There is *one* set of indices (p, q, r) such that $x_p == y_q == z_r$. Then we take that to be the rhyme scheme and no other possibility is considered.
- There is *m>1* sets of indices $(p_1, q_1, r_1), (p_2, q_2, r_2), ..., (p_m, q_m, r_m)$ such that for all i up to m, $x_{p_i} == y_{q_i} == z_{r_i}$. Then all sets are candidate rhyme schemes where each rhyme scheme has a probability of $1/m$.
- There is a p and a q such that $x_p == y_q$, but no r such that $y_q == z_r$. In this case, the rhyme scheme x_p has a probability of two thirds, while every rhyme scheme z_i has the probability of one third each.

An iterative implementation makes this process simple. We want to find and save the most popular rhyme scheme among all poems and remove all sets of possibilities that contain it. We repeat on the remaining poems over and over until we have no more poems.

Algorithm BLAH shows the pseudo-code for this implementation.

1. rhyme_scheme_counts = {}
2. possible_rhyme_scheme_lists = [poem.rhyme_schemes for poem in poems]
3. all_possible_rhyme_schemes = flatten(possible_rhyme_schemes)
4. most_popular_rhyme_schemes = most_common(all_possible_rhyme_schemes)
5. for each rhyme_scheme in most_popular_rhyme_schemes
6. rhyme_scheme_counts[most_popular_rhyme_scheme)] = all_possible_rhyme_schemes.count(rhyme_sche
7. for each possible_rhyme_scheme_list in possible_rhyme_scheme_lists
8. if possible_rhyme_scheme_list contains any of most_popular_rhyme_schemes
9. remove possible_rhyme_scheme_list from possible_rhyme_scheme_lists
10. if possible_rhyme_scheme_lists is not empty, go to 3.

11. return rhyme_scheme_counts

4.3.4 Filter Below Threshold

This algorithm deals mostly with content-based features such as n-grams, persona relations and types of persona. These features are varied, unpredictable, error-prone and highly sensitive to bias in the corpus. However, they can still provide very interesting and useful data to guide generation and create authentic poems. For example, starting a limerick with 'There was a...' and talking about nature in Haikus.

In this case, we perform the most suitable of the three aforementioned algorithms to collect all possible results. We then apply a filter so that only results that occur with *significant frequency* can be taken into account.

Unfortunately, there is no way to tell what frequency can really be counted as significant. It will vary greatly between types and subcategories of poems and is prone to many red-herrings. Trial and error has given the optimal threshold of 9% of the corpus, but this will need further research to refine.

4.4 Testing and Validation by Inspection

We test and validate this phase by looking at two factors:

1. Accuracy of our assumptions and predictions made at the start of this chapter.
2. Comparison of results with known poetry theory.

While some automated tests can be written to make sure that the algorithms above are implemented correctly, there is no desired output against which to compare. We cannot check for exact matches with poetry theory because this is an *investigation* where known theory is the hypothesis. Writing tests that fail if the hypothesis is not correct is a very biased investigation to say the least.

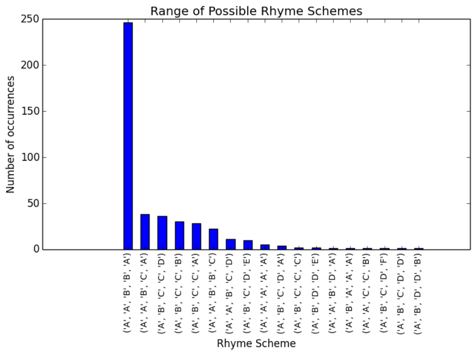


Figure 4.3: AABBA rhyme scheme by far the most common

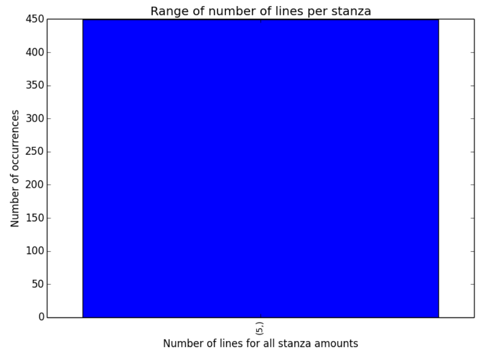
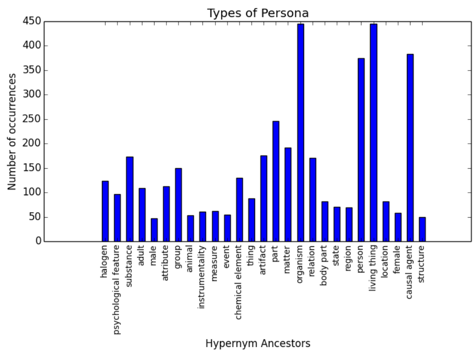
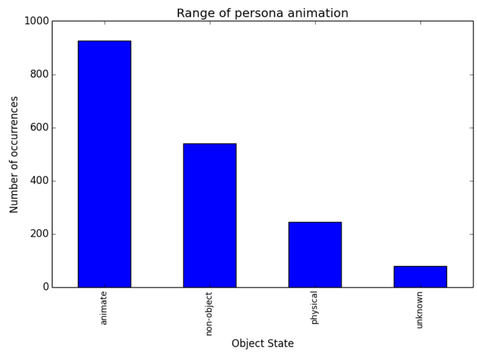


Figure 4.4: Five lines long, one stanza



(a) Limericks talk a lot about people



(b) Most persona are either animate objects or 'concepts' such as places

Figure 4.5: Limericks generally talk about people and other animate objects

Therefore, the best way to test and validate the results of this stage is by inspection. Python provides a convenient interface for graphing large sets of data, which also make it easy to inspect by eye.

The corpus used to test the system comprises of 450 limericks scrounged from the Internet. As mentioned earlier, poems gathered from the Internet are likely to have more outliers and anomalies, which makes for a more rigorous examination of this system.

A noteworthy subset of the results are shown in Figures 4.3 through 4.9. The y-axis is given in raw values instead of probabilities for this testing stage.

The results of these and other corpora will be explored fully in the Evaluation chapter later. It is sufficient for the moment to say that these are good results because they

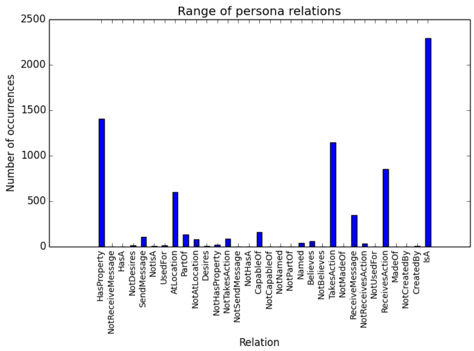


Figure 4.6: Limericks talk about who these people are, what properties they have, what they are named, where they are, their capabilities, what they do, what is done to them etc.

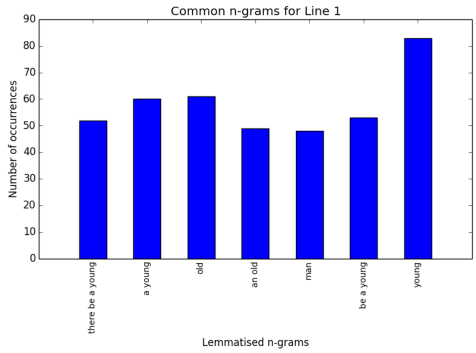


Figure 4.7: The first line often introduces the person, usually as either old or young

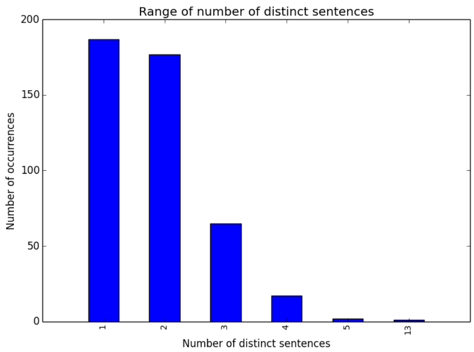


Figure 4.8: Whole poem seems to be either one or two sentences. This actually tells us that full stops are not needed, but if they are used then the whole poem is generally two sentences long.

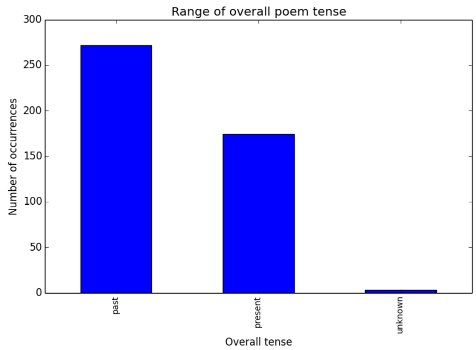


Figure 4.9: Never in future tense, mostly in past an sometimes in present

match sufficiently to known poetry theory. They are also usable in the generation phase to apply rules but also guides us on the strength of these rules, suggesting where divergence may be applicable as desired.

4.5 Concurrency and Performance

Since poems are analysed and features are aggregated *in isolation*, there is plenty of scope for concurrency.

For the poem analysis, a thread pool managed the concurrent execution of 10 workers, each analysing one poem. Trial and error showed 10 to be the optimal number of worker threads. Large resources such as the CMU Pronouncing Dictionary and WordNet were created once and shared amongst the threads.

Under this set up with machine specifications as given in the Appendix section A.4, 450 limericks were analysed in an average of 2 hours and 14 minutes.

This was sufficiently fast for this corpus and since this is data is pre-processed and stored. However, using Python's Pool Executor library rather than the Threading module enables us to move over to processes instead of threads with minimal effort if we need multi-core parallelism. It should be noted that this might be slower due to larger overhead and the need for large resources to be created in multiple locations since memory cannot be shared.

The generalisation process was actually fastest without the overhead of constructing the thread pool. With a thread pool of any number of workers, the average processing time is above 42 seconds for the results of the 450 analysed limericks. Without threading, the process took 40.1 seconds on average.

This generalisation process is fast enough for it to be done over again with different given values as discussed in Section 4.2 earlier in the chapter. This is will be very important in getting precise guidance during the generalisation phase.

Chapter 5

Poetry Generation

The final stage of development entails generating new, novel and creative poetry by utilising the information gathered in the previous analysis and interpretation sections.

5.1 Approach

Most previous attempts at automatic poetry generation, including ones mentioned in Section 2.2, prioritise adherence to poetic features and structure above all else. This is done with the justification that:

- poetry rarely follows syntactical rules of English grammar.
- readers of poems, as humans, are extremely apt at finding subtle meaning in language, even if it was not intended by the author.

However, grammatical rules in poetry are broken for some poetic purpose like matching a rhyme scheme or rhythm pattern. They are therefore broken with precision and intention, not arbitrarily or randomly.

Furthermore, we discussed in Section 2.4.3 that even grammatically correct sentences can be completely nonsensical. We require understanding of the semantic relationships between words to be able to write with intention and in a way that can be understood by the reader.

As discussed in Section 2.1.1, the purpose of poetry is to deliver a specific, intentional message. This cannot be done without control over language both syntactically and semantically.

On a separate note, this implementation should be able to produce any type of poem accurately to the information gathered in the analysis and interpretation. We cannot make any assumptions on the length of the lines, the existence of rhyme or the topics covered by the poems.

Therefore, our approach will be **content first**. We aim to produce poetry that follow syntax and semantics as far as possible and only make specific exceptions for the sake of poetic features.

5.2 Generating Simple Sentences

We attempted to recognise the existence of ConceptNet-style relations during the analysis phase. Our interpretation of the results generalises the use of these relations, so it is natural that we use them to guide generation. For example if we intend to write a poem describing a woman named Mary who wants a monkey, we would start with the persona relation hubs in Figure BLAH below.

1-Named-¿Mary 1-IsA-¿woman 1-TA-¿chase 2-IsA-¿monkey 2-RA-¿chase

These relations outline the context of the poem. In lieu with our *content first* approach, we begin by directly translating these relations into syntactically correct natural language *clauses*, which can be organised into lines in a poem. We make use of two tools to do this; **SimpleNLG**[REF] and **FrameNet**[REF].

5.2.1 SimpleNLG

SimpleNLG is a Java library that provides useful functionality for natural language generation using the ideas of Reiter and Dale described in Section 2.4.6. In particular, it enables us to build *phrases* for nouns, verbs, prepositions, adjectives and adverbs that can be enriched with grammatical metadata such as tense, aspect and perspective (for verbs), as well as plurality, gender and animation for nouns.

These phrases can then be put together into *clauses* that define the roles of each phrase in the desired sentence, for example by specifying the subject and object noun phrases. It then *realises* a grammatically correct natural language sentence taking all of the provided information into account.

We choose to use SimpleNLG because, as its name suggests, it is very simple to use. It provides an almost bespoke interface into the features of the sentence that we picked out in the analysis stage, allowing us to apply the results directly to our new poems. Despite it being a Java library we are able to execute it and access its objects from Python via JPype[REF], a tool for running Java libraries accessing the Java Virtual Machine from Python.

5.2.2 FrameNet

As described in Section 2.4.3, we can use FrameNet to *constrain* the building of clauses around individual words. We used it when deriving relations with SEMAFOR in Section BLAH by checking for the existence of certain frames. Now we reverse the process; use selected frames that correspond to particular relations to guide the type and roles of phrases during generation.

Take the Desire relation for example. The frame in FrameNet that would correspond to this is the Desiring frame, which has a list of words, referred to as *lexical units* or *LUs* - that are used in statements that indicate desire, e.g. want, lust, yearn etc. Each of these LUs come with their own *lexical entries*, which describe the frame elements and importantly *Valence patterns*. These define the types of phrases that can be built around a particular LU.

Take the *yearn* LU. It's Valence Patterns are shown in Table BLAH:

Table: https://framenet2.icsi.berkeley.edu/fnReports/data/lu/lu6599.xml?mode=lexentry

They are grouped by the permutation of frame elements, e.g. Event+Experiencer vs Experiencer+Focal_participant. Each group comes with one or more Valence Patterns that indicate the type of phrase (NP means noun phrase, VP means verb phrase etc.), as well as the role in the clause - Ext (Subject), Obj (Object) and Dep (Dependency or Indirect Object).

5.2.3 Translating Relations into Clauses

The general translation process can be seen in Figure/Algorithm BLAH.

Figure:relation -¿ Get corresponding Frame -¿ Look up LU -¿ Select a Valence Pattern

-¿ Order Phrases - ¿ Fill in Gaps

Suppose we take the first relation in Figure BLAH above. We have a character whose ID is 1 and has the name 'Mary'. The corresponding frame for the 'Named' relation is *Referring_by_name*.

However, it is not quite a perfect match because we want a verbs only, whereas this frame includes LUs of other parts of speech as well. Each LU gives indication of its part-of-speech (POS) so we can filter by that.

Another complication is that the verb 'to name' does not exist in the FrameNet lexicon at all. The closest match is the adjective 'named' found in the *Being_named* frame. So we need to be able to look up this LU manually as well.

Yet another issue is that not all of these LUs come with full and accurate Valence patterns. FrameNet is an ongoing project with more data being added continuously, but we do not want to use a word that does not have an accurate set of Valence patterns. Thankfully, the status of every LU is provided, so we only look up those that have the *Finished_Initial* annotation.

It is important to note that all relation translations may have their own set of similar caveats that are to be considered on a case by case basis to improve the quality of lines generated.

Back to our example, after all filtering we are left with two LUs: *call.v* and *named.a* where the string after the full stop in the LU indicates the word POS. We choose one randomly with equal weights. We use *call.v* for this example.

Figure: https://framenet2.icsi.berkeley.edu/fnReports/data/lu/lu11210.xml?mode=lexentry

This LU has three groups of Valence patterns, as can been seen in Figure BLAH. Each group of patterns has an *TOTAL* count, indicating the occurrence frequency of this pattern in their annotated corpus. We assume that these are representative for our purpose and choose the group with the highest total occurrences.

Each pattern within a group also includes an occurrence count. However, the phrases produced by the system would be predictable and mundane if we always choose the most popular again. Instead we select randomly, weighted for occurrence scores. Let our selected Valence pattern be the third one from the top in Figure BLAH.

We then order the phrases according to how they would appear in a sentence using

Algorithm Blah.

[Algorithm BLAH for sorting]

The Valence pattern does not include the LU itself so we add as a verb phrase immediately after the subject.

Finally we fill in the gaps by adding carefully selected words to each phrase. We discuss word selection in Sections 5.3, 5.4 and 5.6.2.

5.3 Semantic Network of Common Sense

We talked about the idea of modelling the mind as a search space by connecting related *concepts* together in a network in Section BLAHINTHEBG. We gathered inspiration from ConceptNet and discussed Tom De Smedt's attempt to model creativity with such a construct in Sections BLAHINTHEBG and BLAHINTHEBG.

We aim to replicate such a construct bespoke to our purposes, outlined in this section.

5.3.1 The Network

The primary purpose of our network is to help us build clauses out of one or two words. For this, we require our semantic network to give indications of **actions** that nouns take and receive, as well as properties they have.

The secondary purpose of the network is to provide associative data so to maintain a cohesive topic flow through the poem, rather than jumping between a variety of topics.

5.3.1.1 Nodes

As with previous attempts, the nodes of the network are the concepts represented by words. However, we enrich these nodes with the POS of the word so to disambiguate between different meanings (e.g. *bear* the noun and *bear* the verb).

5.3.1.2 Edges

The edges of the network indicate the relationship between concepts. The edges are directed and any that can be reversed are done so manually when they are added to the graph. The types of edges are:

HasProperty

Stereotypical descriptions of the head of the relation.
doctor.n - HasProperty → smart.a
run.v - HasProperty → quickly.adv

IsA

Taxonomy of the head of the relation (Hypernymy).
E.g. banana.n - IsA → fruit.n

PartOf

The head of the relation typically a constituent or member of the tail.
finger.n - PartOf → hand.n

TakesAction

The head of the relation typically performs the action in the tail.
lion.n - TakesAction → roar.v

ReceivesAction

The head of the relation typically has the action in the tail done unto it.
book.n - ReceivesAction → read.v

RelatedTo

General, non-specific, reversible association
fat.a - ReceivesAction - eat.v
eat.v - ReceivesAction - pizza.n

All edges are of equal weight. However, there is potential for prioritising certain types of relations or between concepts. For example, we could weight scientific words higher than regular ones to make the program lean towards a more advanced vocabulary.

5.3.1.3 Concept Halos and Fields

The Concept Halo and Field work in the same way as described in Section BLAHINTHEBG. We use them to fill in the gaps in the sentence with intelligent word selection. For example, if we are given a verb, we can lookup the Concept Field for that verb looking only at TakesAction and ReceivesAction edges to find possible subjects and objects for the verb respectfully. Similarly, we can find modifiers for verbs and nouns by looking up HasProperty edges their respective Concept Halos.

5.3.2 Sources

There are various sources for semantic relationships between words. Some are more applicable to our use case given the nature of relations that we desire and the quality of the source.

There are many sources that we have not used only due to the scope of this project. We discuss them in Section BLAHINTHEEVAL.

5.3.2.1 Collocations

The Oxford Collocations Dictionary[ref] is a high-quality source of word combinations. For any given word, it provides the set of words and POS commonly occur in relation to it.

The dictionary entry for *custard* can be seen in Figure BLAH. It gives common adjectives that are used to describe custard, which can clearly be converted into HasProperty Relations. It also shows that custard receives the action of being made, poured and strained, while it takes the actions of thickening and setting. Further, it is closely related to powder and pie.

We can extract relations directly by parsing this dictionary, which is freely available online in compressed HTML format. As the dictionary is developed by Oxford and is used by students of English, its entries are very high quality and dependable.

5.3.2.2 Associations

The University of South Florida Free Associated Norms[ref] provides associations collected directly from human participants. It is used by the Department of Psychology, and is therefore another high quality source of associated words that can be depended on for quality as it was collected in a scientifically sound manner.

The POS of each word is provided but no direction in the association can be determined. Therefore, we may be able to assume that a noun has the property of an associated adjective and that an adjective is a property of associated nouns. However, we cannot assume the whether a noun takes or receives the action of an associated verb. Therefore we use the more general 'RelatedTo' relationship instead.

5.3.2.3 NodeBox Perception

The NodeBox Perception model includes the data used in De Smedt's attempt to model common sense as a network, as discussed in Section BLAHINTHEBG. There is plenty of overlap in the types of semantic relations used in Perception as we will be using for this project, including 'is-a' (IsA), 'is-property-of' (reversed HasProperty), 'is-part-of' (PartOf) and 'is-related-to' (RelatedTo). We generalise all other relations to our *RelatedTo* relation.

This data was manually entered into the system and continues to be used for research, which means that it is a high-quality source of data.

5.3.2.4 WordNet

WordNet is a widely used, high quality lexical database that provides data on hypernymy (IsA relations) and meronymy (PartOf relations). However, since WordNet is very large and already has fast and simple interfaces from Python, we do not merge it with our Knowledge graph at this point. While it may prove useful to consolidate all data in a single graph in the future, it is not necessary at this stage as we can still benefit fully from its data.

5.3.2.5 Google Search Suggestions

Tony Veale's attempt at modelling metaphors[ref] introduced a clever use of Google's search suggestions as a method of tapping in to endings of a sentence. While we currently do not preprocess relations in the way he did to produce Metaphor Magnet, we do use a similar method to help complete sentences, particularly those with indirect objects.

For example, we would be able to produce a sentence like 'Mary hit the nail with' but be unable to complete the sentence due to a lack of information of indirect objects. However, Google suggestions auto-complete 'hit the nail with' with 'hammer' as one might expect.

This is the least dependable source of information that we use, so we only use it for that specific case and do not add any to our knowledge network.

5.3.3 Concept Similarity

Similarity between concepts in the knowledge network is the primary method of guiding word choice throughout the poem. We deal with two main types of similarity; associative and symbolic.

5.3.3.1 Associative Similarity

Associative similarity between concepts is represented by the length of the shortest past between them in our network. We use Dijkstra's Algorithm[ref] as the basis of finding the shortest path.

Associative similarity helps us choose the best replacement word from a list of candidates. This is very useful when rephrasing to match rhyme and rhythm and will be explained in greater detail in Section 5.7.

It can also help us choose applicable words to start new lines of poetry that stay within context of words used in surrounding lines. This is explained in Section 5.6.

In both of these cases, we are finding the *most* similar concept from a list of candidates of unknown length. This can quickly increase both space and memory complexity without any pruning. We therefore alter Dijkstra's Algorithm to have a limited depth. This way, if we know the shortest path to one of the candidates is x, we can abort any further searches that go beyond length x for the shortest path.

5.3.3.2 Similarity Paths

Paths from one concept do more than show the extent of the relationship - it also shows intermediary concepts taken to get from one to the other. This can be a powerful way of joining two concepts together that may not be on consecutive lines.

For example, suppose the first line of a poem mentions the concept *'man.n'* and the third line *'myth.n'* (Section 5.6.1 explains how this could happen). The path in the knowledge network between these two concepts goes via the concept *'mystery.n'*.

5.3.3.3 Symbolic Similarity

Symbolic similarity can be attained through inductive reasoning and substitution by concepts that have a very similar concept halo and field up to a certain depth. Simply put, we can find similarity with the 'Duck Test'; if it looks like a duck, swims like a duck and quacks like a duck, it probably is a duck. Object-oriented programmers can relate this idea to the Liskov substitution principle[ref].

Symbolic similarity has many use cases, including the new method of anaphora resolution proposed in Section BLAHINANALYSIS of the Analysis phase. For this phase, it enables us to implement the following poetic features:

Similes

We can describe a property or an action of a persona in our poem by comparing the persona to an concept in our knowledge network that (stereotypically) has that property or takes/receives that action.
The relation *1-TA*$\rightarrow$*run quickly* can be built into the phrase 'runs like a cheetah' by recognising that cheetahs are stereotypically quick runners.

Metaphors

Similarly to Similes, we can find concepts in our network that share a number of semantic relations with the persona we are trying to describe.

Any persona with relations *1-TakesAction→crawl*, *1-HasProperty→social*, *1-HasProperty→small* is analogous to an insect because it's concept in the graph shares those relations.

Personification

If we are given an inanimate object as a persona, we can describe it as if it were sentient by looking giving it relations that belong to a sentient being.
A car could be personified by giving it all the relations that a lion has in our concept network, to produce sentences like *'The car roared'*.

Irony

Our knowledge network is made up of semantic attributes that are expected for each concept. Irony can be attained by producing lines of poetry that describe a concept doing the opposite of the concepts given in our graph.

5.4 Persona Creation and Management

5.4.1 Referencing Specific Persona

When we convert the relation *1-Named→Mary* in Figure BLAH to a phrase, we need to refer to the persona using IsA relations. So when we build the clause with the verb 'named' and object 'Mary', we will look up the persona that this relation was taken from and find an IsA relation. In this example, we will find woman, and when we add the determiner 'a', we get the clause *'a woman named Mary'*.

Now when we convert the *1-TA→chase* relation, we follow a similar process of finding a way to reference character 1, with the inclusion of the name of the persona (if available) as another candidate for the reference. This will give us either *'The woman chases'* or *'Mary chases'*.

5.4.2 Adding New Persona

Now since the verb 'chase' is *transitive*, i.e. it requires a subject **and** an object, we need a persona that receives the action 'chase'. If we look through the relations of the other known persona, we will find the relation *2-RA→chase* and we use *2-IsA→monkey* to derive the corresponding reference to that character. So we can complete the sentence by adding *'monkey'* to the end, giving us either *'The woman chases a monkey'* or *'Mary chases a monkey'*.

Suppose we could not find a corresponding ReceivesAction relation for 'chase' in any of the current persona. We must then find a concept in our knowledge network that receives the action 'chase'. We do this by looking at the concept field of the *'chase.v'* concept in our knowledge graph for only the edges with type ReceivesAction. This gives us the required list of concepts from which to select the object of the clause. We use concept similarity to find the most applicable object given the current context of the poem; a process to be described in detail in Section 5.6.2, but for now let's assume that we use the concept *'thief.n'*.

When we add a new noun to the poem, we need to create a new persona object for it so that we can refer back to it later. The new character is instantiated with an IsA relation to its noun word, i.e. *thief*. Its plurality, gender and animation are all derived from the word itself using the process described in Section BLAHINTHEANALYSIS.

5.4.3 Anaphora and Presupposition

So far we are only able to refer to persona by Named or IsA relations that are already part of the persona's relations. This leads to to combinations of clauses such as *'A woman named Mary. Mary chased the monkey.'*, which is completely *un*natural language.

The natural way to phrase it would be *'A woman named Mary. She chased the monkey.'*. In this case, the introduction of the pronoun 'she' is a contextual reference back to the aforementioned persona. We discussed the resolution of such anaphora in Section BLAHINTHEANALYSIS.

Another option for variety in reference is to use *presuppositions*; implicit assumptions about context. This can give us clause combinations such as *'A woman named Mary. She chased the monkey. The animal stole her banana'*. Here we refer back to the monkey persona by looking for a hypernyms in WordNet and IsA relations in our knowledge network for the concept *'monkey.n'*. The implicit assumption made is that monkeys are animals, so the reader would understand the reference.

A more advanced method of presupposition introduction would be to look at stereotypical recipients (concept field) of the concept *'chase.v'* in our knowledge network. This may give us the clause combination *'A woman named Mary. She chased the monkey. The thief stole her banana'*. In this case, we are making an implicit assumption that the monkey is a thief because, according to our knowledge network, thieves stereotypically get chased. This assumption gets added to the context and can be used to develop the story, a process described in Section 5.6.

5.4.4 Semantic Types

Suppose we had the relations *2-RA→chase* and *2-IsA→monkey* but with no corresponding TakesAction relation in any other known persona. We would need either need to create a new persona to be the subject of the clause or we would choose an already existing persona to be the subject.

In the former case we can simply look up our knowledge network in the same way we did when finding the object of the sentence. However, if we run across this case often (which we might, see Section 5.6.2), we could end up introducing a new persona every line which can lead to a much less cohesive poem. So we would like to use the latter case and look at reusing persona that we already know about.

There's a trap here though. What if the only other persona we knew of was an inanimate object like a plate? Or a non-object like evolution? To say *'A plate chases the monkey'* or *'Evolution chases the monkey'* makes no sense. Unless we are using personification, we would want a sentient being to be the subject of the 'chase' action.

Thankfully, FrameNet provides *semantic types* with every frame element of a frame. This indicates the required animation of any noun in all Valence patterns of the LU. We also store the animation of every persona in the system upon creation, so we can refer to this when deciding if a persona is a relevant match with a particular semantic type. If no existing persona are applicable, we can revert back to finding one from our knowledge network.

5.5 Poem Initialisation

Before we start building any poem, we need to construct the basic structure for the desired form of poetry. We also want to be able to take some *inspiration* to initialise the content that can either be sampled from the knowledge network or from the user.

5.5.1 Overall Structure

Our new poem first needs to be initialised with values of features that span across the entire poem, namely:

- Number of stanzas
- Number of lines per stanza
- Number and locations of repeated lines
- Tense
- Perspective
- Rhyme Scheme

All of these can be retrieved directly from the template developed in the previous chapter. The template is made up of probability distributions for the various values a feature can take on, so by default we get the value by sampling this distribution.

However, if one result is clearly more probably than the rest, we want to treat it as unambiguous. For example, more than half of the rhyme schemes found for limericks were AABBA, as pictured in figure 4.3. Each of the other values are some slight variation on AABBA (e.g. AABCA, ABCCB etc.) and occur less than a ninth of the time at maximum.

To generalise this, we take the rule that if the any feature has a single value in its probability distribution that occurs more than half of the time and the next highest value is less than a third of that value, we treat it as unambiguous and *always* apply it to the initial structure of this type of poem.

5.5.2 Inspiration

The poetry generation process can be seeded *inspiration*, which can be in the form of words or relations. Inspiration can come from the user, or the program can come up with it itself. Inspiration from the user is input manually into the system.

When the program derives its own inspiration, it looks to the common hypernym ancestors of previous poems found in section BLAHINTHEINTERPRETATION. It chooses a random hypernym and recursively looks for all holonyms in WordNet recursively, out of which one will be randomly selected. A persona is then created and used as the single source of inspiration for the rest of the poem. Section 5.6.1 explains how.

5.6 Incremental Growth

To ensure some level of cohesion between lines in the poem, we will build line by line, choosing contextually relevant words and relations on the fly.

5.6.1 Applying Inspiration

As we know, new lines are created by taking a relation from a persona and building it using the process described in Section 5.2. If these relations are provided in the inspiration, we first allocate them to lines in the poem based on two factors; relation type and rhyme scheme.

The template derived from the previous parts of the implementation tracks relation usage by line. We use this to decide the order that relations should appear. For example, the Named relation is most likely to appear in the first than any other line in a limerick, as shown in Figure BLAH. Therefore, if any Named relations exist among any persona, that relation comes first.

Rephrasing statements occurs when we need to adhere to a poetic feature such as rhyme or rhythm, a process described in Section 5.7. However, we want to stay as true to the initial inspiration as possible so we want to avoid rephrasing any clauses built from relations provided by the user whenever we can.

Therefore we apply the simple heuristic that relations provided by the user should fall on the first line that introduces a new rhyme token. These lines automatically bypass rephrasing for rhyme adherence. Naturally, this is only possible when the number of provided relations is less than or equal to the number of rhyme tokens so it is not always possible, but it provides the user with a useful guide on how many relations to input.

This ability to allocate relations to certain lines also provides flexibility of control for the user. If they *want* a relation to mark the ending of the poem, they can configure it to do so.

Once relations have been allocated to lines we can begin composing lines in natural language.

5.6.2 Creating New Lines

New lines are created by taking a relation from a persona and building it using the process described in Section 5.2. Even though inspiration may not be distributed consecutively through the poem, we always start with the first line and build linearly down to the last line.

Suppose we are building limericks with an *AABBA* rhyme scheme and that we are provided with one relation by the user, providing content for line 1 only. Line 1 would be built first without any rephrasing for rhyme. The process of building the rest of the lines are dependant on whether or not they should rhyme with a previous one.

5.6.2.1 Rhyming Lines

When the second line starts being built, we recognise that it is has not been allocated any relation so we must create one. We also know that it needs to rhyme with the first line so we can avoid rephrasing again by seeding the line with a word that rhymes, and then building the rest of the line backwards from it.

The process of finding rhyming words is outlined in Section 5.7.1.1. We select one within context (as described in 5.6.3), and build a relation according to its POS to guarantee that it ends up at the end of the clause once it is built.

For example, if the rhyming word is an adjective we cannot build a TakesAction phrase from it because those phrases end in either a noun or a verb when created. We could rephrase for rhyme later but that would go against our 'context first' approach. The possible actions for each POS of the chosen rhyming word is as follows:

Noun

Verb

Adjective

Adverb

Once a relation created of the chosen type and with the chosen rhyme word, it is added to the most applicable persona based on semantic type and associative similarity (see Sections 5.4.4 and 5.3.3.1). This process is followed when building the fourth and fifth line of our limerick as well.

5.6.2.2 Blank Lines

The third line has no allocated relation and is the start of a new rhyme token. In this case we can add any relation or word we want as long as it stays within context(Section 5.6.3).

We therefore choose any word that is associatively similar to those in context and then build a relation based on its POS as in the previous section.

5.6.3 Keeping in Context

We choose words based on context to achieve coherent topic flow throughout the poem. In our implementation, we define context as **the concepts mentioned in immediately surrounding lines**, i.e. the context of line 2 is determined by the words used in line 1 and the head and tail of the relation that is expected to be translated in line 3.

Whenever there is a choice between using any set of candidate concepts, we choose the one that is most associatively similar (as defined is 5.3.3.1) to the set of context concepts. For example, if we had the concepts *'man.n'* and *'pizza.n'* in our set of context concepts and *'fat.a'* and *'cat.n'* as our candidate concepts, we would choose to use *'fat.a'* because it is more associatively similar to *'man.n'* and *'pizza.n'*.

5.7 Rephrase for Poetic Features

In lieu with our *content first* approach, we want to avoid rephrasing as much as possible. However, a poem is not a poem unless it adheres to rules of poeticness such as rhyme and rhythm. Therefore, rephrasing a clause will still happen frequently. The challenge is in keeping the phrase as true to the original as possible.

The main two cases where rephrasing is required is to match with rhyme and rhythm. We will look at the full tactics of each.

5.7.1 Rhyme

Rhyme is a very powerful poetic tool that comes naturally to many human authors. If a poem doesn't rhyme when it is supposed to, it is a big give-away that it may not have been created by a human at all. So while we will try to stay true to the original, it is vital that the poem rhymes when it is supposed to at any cost.

5.7.1.1 Finding Rhyming Words

Rhyming words are retrieved from the RhymeBrain API[ref]. This tool accepts *any target string of letters* (not just words from a dictionary) and finds rhyming words. This works well with our high priority for strong rhyme because it rarely fails to give back rhyming words and tends to return more possibilities than alternatives like Wordnik[ref]. It also means that even the most obscure names or other words input from the user can be used effectively by the system.

Results from the API also come with a score, allowing us to balance strength of rhyme with applicability to content. For example, if we are looking to rhyme with the target word *'address'*, we get the results shown in Table BLAH.

'Misaddress' is the only word with the top rhyme score. However, if *'access'* is highly applicable in context, we may choose that instead despite the lower score. We do not consider any words lower than a score of 200.

The main drawback of the RhymeBrain API is that it tries to find rhyming words with very strict rhyme; at least as many syllables as the target string with exactly matching emphasis. This is why ***'dress'*** and *'im**press**'* are not suggested as candidate rhymes for *'**add**ress'*.

To counter this, we try to find the rhymes of a suffix of the target string that most probably only contains the last syllable. To do this, we convert the word into its pronunciations as in Section BLAHINTHEANALYSIS. We then find the first consonant phoneme from the right with a vowel phoneme to its left. We then search for the probable letters in the original string corresponding to that consonant phoneme and take the suffix from that letter onwards.

For example, the word *'address'* has the pronunciations *['AH0', 'D', 'R', 'EH1', 'S']* and *['AE1', 'D', 'R', 'EH2', 'S']*. In both cases, we select the 'R' phoneme and match it up to the 'r' in 'address', giving us the suffix 'ress'. The results for words rhyming with 'ress' are shown in Table BLAH.

This gives us many words that rhyme with 'address', including *'dress'* and *'impress'*.

5.7.1.2 Guaranteeing Rhyme

The general tactic is to only replace words with synonyms or very close associative similarity. For example, I could exchange the word 'horse' with 'pony' if I needed to rhyme with 'Tony'.

If that is not possible, we look to *extend* the clause by adding applicable adjectives or adverbs to the end of the sentence. SimpleNLG enables us to do this by adding *post modifiers* to noun and verb phrases. These are words that will appear immediately after the parent phrase.

We first find the final word in the phrase of the clause that is being rephrased. If it is a noun, we filter out any non-adjective candidate rhymes. Similarly, if it is a verb we filter out non-adverbs. We then look up the concept halo for the target word, looking only at 'HasProperty' edges. This gives us a list of words that are commonly used to describe the target word. We then compare this list to the candidate rhyming words, find the most associatively similar pair and add it as a post modifier to the phrase of the target word.

This process guarantees that the final word in the clause will have the necessary rhyme when realised, but also keeps the added or replaced word very close to the original statement.

5.7.2 Rhythm

Rhythm is much less stringent than rhyme because most words can be mispronounced without changing their meaning. For example, in the fifth line of Shakespeare's Sonnet 116 he pronounces the monosyllabic word *'fixed'* as *'fix**ed**'*, essentially adding a syllable and changing the stress.

Therefore, we stick to our *content first* approach and only try to find a best match for rhythm without compromising content.

5.7.2.1 Syllabic

Syllabic rhythm, as described in Section BLAHINTHEBG, defines the number of syllables each line should have. After we have created phrases for the line and matched any rhyme scheme correctly, we can extend or reduce the syllable length of the line to match syllabic rhythm.

5.7.2.1.1 Extending If the number of syllables in the line is fewer than the required amount, we must add syllables to the line. The simplest way to do this is to add more modifiers.

We randomly choose a phrase from those that make up the line. We then look up the knowledge network for modifiers by looking the target word's concept halo for the 'HasProperty' relation. We randomly select one and add it as a modifier to the phrase of the target word as long as does not add more syllables than required to make up the shortage.

We repeat this process, expanding the depth of the halo to incorporate more words when necessary, until we have exactly the right number of syllables.

SimpleNLG chooses where the modifier should appear in the sentence unless explicitly told to where to put them. This can be done by adding it as a *pre-modifier* to the phrase, enforcing the word to appear *before* the target word, or a *post-modifier* to enforce appearance after the target word as discussed earlier.

To ensure that we do not tamper with the rhyming word, we explicitly instruct are only allowed to ever add pre-modifiers to the phrase of the rhyming word.

5.7.2.1.2 Reducing Occasionally the syllabic length of the line exceeds the required number of syllables. In this case, replacement of words is our only option. We utilise our knowledge graph to find replacements with close associative similarity, but there is still a risk of losing coherence. We never replace the rhyming word.

The process involves finding the longest word in the sentence and replacing it with the most applicable word with fewer syllables. We repeat until we either have exactly the right number of syllables or if we overshoot and have too few, we using the extending process to get it back up to the exact required amount.

5.7.2.2 Accentual

Any poem with accentual rhythm has, by definition, syllabic rhythm. Therefore we rephrase to match syllabic rhythm first before we fit to accentual rhythm. That way we know that we have exactly the right number of syllables at the beginning of the rephrasing process.

Let's take *'Woman chased the hairy monkey'* as our line and '01001001' as our required stress pattern. If we put one on top of the other as in Figure BLAH, we can pair up each word with its required pattern.

Start from the rightmost word, 'monkey'. Since we do not want to tamper with rhyme, we disregard this pattern-word pair. Moving along to the left we get the word 'hairy', which has the required pattern of '10'. This matches with the actual stress pattern for 'hairy', so nothing needs to be done and move along again. The same happens for the words 'the' and 'chased' because they are both monosyllabic and can be either stressed or unstressed.

Finally. the word 'woman' is paired with the stress pattern '01'. We have a mismatch because, according to the CMU Pronunciation Dictionary, the stress pattern for 'woman' is '10'. We need to replace the word.

There are three approaches to this problem:

1. The simple and naive method would be to choose any random word that fits the required pattern, but that contradicts the *content first* approach.
2. We could only look at synonyms and hypernyms/IsA relations and replace with any that have the required pattern, but there is a risk that none of them do, in which case we do not replace the word at all, keeping the incorrect pattern.
3. The final approach involves indexing all words in the knowledge network by possible stress patterns. We select the most associatively similar word from the values of the required stress pattern. However, this again leads to the possibility of finding a poor replacement word in terms of preserving the integrity of the content.

We choose to use the second method given our priorities for this project. In our example we would replace the word with 'adult' as it is a hypernym of woman.

Chapter 6

Evaluation Plan

We will employ four methods of evaluating this project.

6.1 Comparison of Analysis Results to Theory

The Analysis and Abstraction phases attempt to discover the common features of any type of poetry. Current theory of these common features have been derived and documented by poets through their own analysis. We wish to investigate whether this system is able to find everything documented that has been documented by poets. It is likely that the system could only find a subset but it will be interesting to see which points were missed. A full comparison will be made and explanation for any missing features will be attempted, as well as possible algorithms to detect them that could have been implemented.

A particularly interesting result would be if the system manages to find a common feature of a type of poetry that poets have not yet found. If even one case of this occurs then this would make a strong case for the ability of computers to understand, interpret and find patterns in natural language. Further investigation would surely be warranted into how much a computer system would be able to find that human readers have overlooked.

Note that this section does not expect the system to infer the effects of any of the poetry features on the reader, just simply detect the use of the features.

6.2 Turing-style Tests

The simplest way to check the quality of poems produced would be to show people a poem either generated by this system or written by a human without telling them and asking them to determine whether the author was man or machine.

However, this is highly dependent on the reader's fluency of English, understanding of poetry and imagination. A poor English speaker with little to no understanding of poetry and with a far reaching imagination could easily be fooled into thinking that a piece of text was written by a human rather than a computer.

Therefore, we propose a survey that asks for these three measures to be told truthfully. It then randomly shows a poem and asks whether it was written by a human or computer. This way, we can get a demographic of those who are fooled and those who are not so that we can see at what level of poetry literacy this system is.

Further, we plan to approach real poets from literary institutions and university departments and ask them to do this survey in person. If they incorrectly believe even one poem to be written by a human and not a computer, then this project is a definite success. For the cases where this does not arise, having an in-person survey will enable us to ask for feedback on what gave us away, which can be used to improve the system and future attempts at poetry generation.

However, Pease and Colton[40] make several arguments that Turing-style tests are not appropriate for poetry generation. We believe that it has its place because ultimately this project is for user consumption as well as research and experimentation. Furthermore, poems attempt to create an emotional connection with the reader, something that cannot be determined other than with a human reader.

Having said that, Colton, Charnley and Pease[18] described the FACE and IDEA Descriptive Models for evaluating computational creativity projects. We believe these models provide a useful evaluation methodology alongside Turing-style tests and so are just as much part of the evaluation of this system as described in Section 6.3 and 6.4. It will also be interesting to evaluate them as an evaluation method as they have only just been proposed.

6.3 FACE Descriptive Model

A full FACE model has four symptoms: examples, concepts, aesthetics and framing information.

- *Examples* will be showcased by the templates generated by the Analysis and Abstraction phase.
- *Concepts* are of the form of the algorithm described for the Generation phase as it takes input from the user or online, the results from the Abstraction phase and several third party libraries to output a poem.
- *Aesthetics* are assessed by running the Analysis phase over the poem again. In fact, this happens several times during the creation of the poem. Any faults are reported back to the next iteration of that poem.
- *Framing Information* is the poem created.

Therefore, we can see that this system should fully abide by the FACE Descriptive Model. We will evaluate the results mathematically as per Colton, Charnley and Pease.

6.4 IDEA Descriptive Model

A full IDEA model has six stages to which the software can reach. We want our software to be in the, fourth or *Discovery stage*.

- *Developmental stage*: this system has a full Abstraction phase to avoid the case that all creative acts undertaken by this system are purely based on inspiring examples. So this system will have surpassed this stage.
- *Fine-tuning stage*: the Abstraction phase only looks for a limited number of overlapping features to provide the template, leading to higher level abstraction. For example, it does not use part-of-speech tags from previous examples or any low level abstractions. We believe the system should be able to surpass this level.

- *Re-invention stage*: the system is able to work off a template provided by the Abstraction phase, but also able to mutate the templates and add or remove restrictions both automatically and guided by the user. Therefore, the creative acts are not restricted only to those that are known and should be able to surpass this stage as well.
- *Discovery stage*: the ability to work off templates derived from Analysis and Abstraction imply that the system is able to generate works that are sufficiently similar to be assessed with current contexts. However, given the flexibility of the mutation and user-guidance ability, it can also produce works that are significantly dissimilar. We believe the system to be able to reach this stage.
- *Disruption and Disorientation stages*: Since templates and constraints on the creative work that are imposed are the results of analysing and abstracting existing works, it is not the case that this system solely produces poetry that is too dissimilar to those known by theory.

Therefore, we can see that this system should reach the desired *Discovery stage* of the IDEA Descriptive Model. We will evaluate the results mathematically as per Colton, Charnley and Pease.

Appendix A

A.1 Dry Run of Generation Phase with Commentary

Input seed: Limerick about a computer that is bored with data and finds poetry fun

1. *There once was (a/an) _____ named ____[A]*
_________________________________[A]
_____________________[B]
_____________________[B]
_________________________________[A]

This is a limerick template straight out of the generation phase.

2. *There once was a computer named ____ [data]*
That was bored with data [data]
It finds poetry fun [fun]
_____________________[fun]
______________________________ [data]

This step introduced the computer, the fact that it is bored with data and finds poetry fun. It does not take any structure into account when adding these.

3. *There once was a computer named Zeta*
That was bored to death with data
It finds poetry fun
___________ sun
___________________ beta

This step filled in the name Zeta since it needed to rhyme with data. It rephrased 'bored' with 'bored to death' to fit the rhythm without losing meaning. Found third line to be complete. Added sun to end of next line since it is an association with fun and follows rhyme scheme. Added beta to end of last line since it rhymes with data (and Zeta)

4. *There once was a computer named Zeta*
That was bored to death with data
It finds poetry fun
Like the summer sun
The revolutionary new system goes into beta

This step added a simile to compare the fun of poetry to the sun since it fit the rhythm structure (alternatively, "like the scorching/setting/sinking sun", but summer sun has alliteration on 'su' rather than just 's'). A phrase was found that ends with beta and was arbitrarily added.

5. *There once was a computer named Zeta*
That was bored to death with data
It finds poetry fun
Like the summer sun
The new system goes into beta

Redundant adjective removed to fit structure in last line. Poem finished.

Note too that we added the input data to the first three lines. Separating them out will make it more coherent, especially in longer poems. Then the associations could be found with surrounding sentences, not just the previous one.

Further rephrasing of lifted sentences, such as the last one, could be beneficial to adding randomness and creativity.

Phoneme	Example	Tranlsation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY LD
Z	zee	Z IY
ZH	seizure	S IY ZH ER

Table 1: ARPAbet phoneme set with corresponding examples

Tag	
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Table 2: Penn Treebank Tagset in alphabetical order

A.2 CMU Pronunciation Dictionary ARPABET Phoneme Set

A.3 Penn Treebank Tagset

A.4 Testing Specifications

- Processor: Intel Core i7-3612QM CPU @ 2.10 GHz
- RAM: 8GB
- Average round trip ping to demo.ark.cs.cmu.edu: 131 milliseconds

A.5 FrameNet Frames to ConceptNet Relations

PartOf

Frame	**Character Frame Element**	**Whole Frame Element**
Architectural_part	Part	Whole
Building_subparts	Building_part	Whole
Clothing_parts	Subpart	Clothing
Observable_body_parts	Body_part	Possessor
Part_edge	Part	Whole
Part_inner_outer	Part	Whole
Part_ordered_segments	Part	Whole
Part_orientational	Part	Whole
Part_piece	Piece	Substance
Part_whole	Part	Whole
Shaped_part	Part	Whole
Vehicle_subpart	Part	Whole
Wholes_and_parts	Part	Whole
Inclusion	Part	Whole

CreatedBy

Frame	Character Frame Element	Creator Frame Element
Creating	Created_Entity	Creator
Cooking_creation	Produced_food	Cook
Intentionally_create	Created_Entity	Creator
Text_creation	Text	Author
Manufacturing	Product	Producer/Factory

MadeOf

Frame	Character Frame Element	Substance Frame Element
Substance	[Noun After]	Substance

Causes

Frame	Character Frame Element	Effect Frame Element
Cause_to_start	Cause	Effect
Causation	Actor	Affected
Causation	Cause	Effect
Condition_Symptom_Relation	Medical_condition	Symptom
Cognitive_connection	Concept_1	Concept_2

Desire

Frame	Character Frame Element	Desirable Frame Element
Desiring	Experiencer	Event
Experiencer_focus	Experiencer	Content

CapableOf

Frame	Character Frame Element	Activity Frame Element
Capability	Entity	Event

UsedFor

MotivatedByGoal

Has

NotHas

Frame	Character Frame Element	Purpose Frame Element
Expend_resource	Resource	Purpose
Using	Instrument	Purpose
Tool_purpose	Tool	Purpose
Ingest_substance	Substance	Purpose
Using_resource	Resource	Purpose
Usefulness	Entity	Purpose

Frame	Character Frame Element	Goal Frame Element
Purpose	Means	Goal/Value

Frame	Character Frame Element	Possession Frame Element
Possession	Owner	Possession
Have_associated	Topical_entity	Entity
Inclusion	Total	Part
Containers	Container	Contents

Frame	Character Frame Element	Possession Frame Element
Used_up	[Character in view]	Resource
Expend_resource	[Character in view]	Resource
Abandonment	Agent	Theme

Frame	Character Frame Element	Name Frame Element
Being_named	Entity	Name

Named

Believes

Frame	**Character Frame Element**	**Belief Frame Element**
Awareness	Cognizer	Content
Certainty	Cognizer	Content
Religious_belief	Believer	Content
Trust	Cognizer	Information
Opinion	Cognizer	Opinion

SendMessage and ReceiveMessage

Frame	**Speaker Character Frame Element**	**Message Frame Element**	**Addresse**
Communication	Communicator	Message/Topic	Addressee
Telling	Speaker	Message	Addressee
Request	Speaker	Message	Addressee
Speak_on_topic	Speaker	Topic	Audience
Statement	Speaker	Message/Topic	Addressee
Prevarication	Speaker	Topic	Addressee
Reporting	Informer	Behaviour/Wrongdoer	Authoritie
Text_creation	Author	Text	Addressee
Chatting	Interlocutor_1	Topic	Interlocuto

A.6 List of collapsable dependencies

- acomp
- advmod
- aux
- cop
- dep
- det
- measure

- neg
- nn
- num
- number
- preconj
- predet
- prep
- pobj
- quantmod

A.7 Onomatopoeia Types to ConceptNet Relations

- laughter: TakesAction → laugh
- laughing: TakesAction → laugh
- hit: ReceivesAction → hit
- hard hit: ReceivesAction → hard hit
- punch: TakesAction → punch
- fall: TakesAction → fall
- light hit: ReceivesAction → light hit
- liquid: HasProperty → wet
- water: HasProperty → wet
- wet: HasProperty → wet
- rain: HasProperty → wet

- coins: IsA → money
- weapon: IsA → weapon
- metal: MadeOf → metal
- rubber: MadeOf → rubber
- music: HasProperty → musical
- disease: HasProperty → ill
- surprise: HasProperty → surprised
- dismay: HasProperty → dismay
- pain: HasProperty → in pain
- telephone: IsA → telephone
- siren: HasA → siren
- electronic: HasProperty → electronic
- static: HasProperty → electronic
- electric: HasProperty → electronic
- television: HasProperty → electronic
- video games: HasProperty → electronic
- horn: HasA → horn
- clock: IsA → clock
- predator: IsA → predator
- animal: IsA → animal
- frog: IsA → frog
- bird: IsA → bird
- cat: IsA → cat
- dog: IsA → dog

References

[1] URL: http://www.wordnik.com/about.

[2] URL: http://www.writtensound.com/about.php.

[3] URL: http://docs.python.org/2/library/difflib.html.

[4] URL: http://www.ark.cs.cmu.edu/.

[5] URL: http://www.ark.cs.cmu.edu/TurboParser.

[6] Collin F Baker, Charles J Fillmore, and John B Lowe. "The berkeley framenet project". In: *Proceedings of the 17th international conference on Computational linguistics-Volume 1*. Association for Computational Linguistics. 1998, pp. 86–90.

[7] Valerio Basile and Johan Bos. "Towards generating text from discourse representation structures". In: *Proceedings of the 13th European Workshop on Natural Language Generation*. Association for Computational Linguistics. 2011, pp. 145–150.

[8] Valerio Basile et al. "Developing a large semantically annotated corpus". In: *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*. Istanbul, Turkey, 2012, pp. 3196–3200.

[9] John Bateman and Michael Zock. *Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003. Chap. 15 Natural Language Generation.

[10] Kim Binsted and Graeme Ritchie. "Computational rules for generating punning riddles". In: *Humor* 10.1 (1997), pp. 25–76.

[11] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'reilly, 2009.

[12] Patrick Blackburn and Johan Bos. "Computational semantics". In: *THEORIA. An International Journal for Theory, History and Foundations of Science* 18.1 (2008).

[13] William Chamberlain and Thomas Etter. "The Policeman's Beard is Half-Constructed: Computer Prose and Poetry". In: *Warner Software/Books, New York* (1984).

[14] Desai Chen et al. "SEMAFOR: Frame argument resolution with log-linear models". In: *Proceedings of the 5th international workshop on semantic evaluation.* Association for Computational Linguistics. 2010, pp. 264–267.

[15] Noam Chomsky. *Syntactic structures.* Walter de Gruyter, 2002.

[16] Simon Colton. "The painting fool: Stories from building an automated painter". In: *Computers and creativity.* Springer, 2012, pp. 3–38.

[17] Simon Colton, Jacob Goodwin, and Tony Veale. "Full face poetry generation". In: *Proceedings of the Third International Conference on Computational Creativity.* 2012, pp. 95–102.

[18] Simon Colton, A Pease, and J Charnley. "Computational creativity theory: The FACE and IDEA descriptive models". In: *Proceedings of the Second International Conference on Computational Creativity.* 2011.

[19] Simon Colton and Geraint A Wiggins. "Computational Creativity: The Final Frontier?" In: *ECAI.* 2012, pp. 21–26.

[20] Jonathan Crowther, Sheila Dignen, and Diana Lea. *Oxford Collocations Dictionary: For Students of English.* Foreign Language Teaching and Research Press, 2003.

[21] Marie-Catherine De Marneffe and Christopher D Manning. "Stanford typed dependencies manual". In: *URL http://nlp. stanford. edu/software/dependencies manual. pdf* (2008).

[22] Tom De Smedt and Walter Daelemans. "Pattern for python". In: *The Journal of Machine Learning Research* 98888 (2012), pp. 2063–2067.

[23] Gustavo Diaz-Jerez. "Composing with Melomics: Delving into the Computational World for Musical Inspiration". In: *Leonardo Music Journal* 21 (2011), pp. 13–14.

[24] Simon Dobrišek, Janez Žibert, and France Mihelič. "Towards the optimal minimization of a pronunciation dictionary model". In: *Text, Speech and Dialogue.* Springer. 2010, pp. 267–274.

[25] George R Doddington et al. "The Automatic Content Extraction (ACE) Program-Tasks, Data, and Evaluation." In: *LREC.* Citeseer. 2004.

[26] Claire Gardent, Eric Kow, et al. "A symbolic approach to near-deterministic surface realisation using tree adjoining grammar". In: *ACL.* Vol. 7. 2007, pp. 328–335.

[27] Albert Gatt and Ehud Reiter. "SimpleNLG: A realisation engine for practical applications". In: *Proceedings of the 12th European Workshop on Natural Language Generation.* Association for Computational Linguistics. 2009, pp. 90–93.

[28] Pablo Gervás. "Wasp: Evaluation of different strategies for the automatic generation of spanish verse". In: *Proceedings of the AISB-00 Symposium on Creative & Cultural Aspects of AI.* 2000, pp. 93–100.

[29] Dan Jurafsky et al. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition.* Vol. 2. MIT Press, 2000.

[30] Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*. 42. Springer, 1993.

[31] Paul Kingsbury and Martha Palmer. "From TreeBank to PropBank." In: *LREC*. Citeseer. 2002.

[32] Peter Kolb. "Disco: A multilingual database of distributionally similar words". In: *Proceedings of KONVENS-2008, Berlin* (2008).

[33] Ray Kurzweil. "Ray kurzweil's cybernetic poet". In: *CyberArt Technologies* (1999).

[34] Hugo Liu and Push Singh. "ConceptNet—a practical commonsense reasoning tool-kit". In: *BT technology journal* 22.4 (2004), pp. 211–226.

[35] Hisar Manurung. "An evolutionary algorithm approach to poetry generation". In: (2004).

[36] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a large annotated corpus of English: The Penn Treebank". In: *Computational linguistics* 19.2 (1993), pp. 313–330.

[37] Adam Meyers et al. "The NomBank project: An interim report". In: *HLT-NAACL 2004 workshop: Frontiers in corpus annotation*. 2004, pp. 24–31.

[38] George A Miller. "WordNet: a lexical database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[39] Douglas L Nelson, Cathy L McEvoy, and Thomas A Schreiber. "The University of South Florida free association, rhyme, and word fragment norms". In: *Behavior Research Methods, Instruments, & Computers* 36.3 (2004), pp. 402–407.

[40] Alison Pease and Simon Colton. "On impact and evaluation in computational creativity: A discussion of the Turing test and an alternative proposal". In: *Proceedings of the AISB*. Vol. 11. 2011, pp. 1–8.

[41] Ehud Reiter and Robert Dale. *Building natural language generation systems.* Vol. 152. MIT Press, 2000.

[42] Karin Kipper Schuler. "VerbNet: A broad-coverage, comprehensive verb lexicon". In: (2005).

[43] Jukka M Toivanen, Matti Järvisalo, and Hannu Toivonen. "Harnessing Constraint Programming for Poetry Composition". In: *Proceedings of the Fourth International Conference on Computational Creativity.* 2013, p. 160.

[44] Tony Veale. "Less Rhyme, More Reason: Knowledge-based Poetry Generation with Feeling, Insight and Wit". In: *Proceedings of the Fourth International Conference on Computational Creativity.* 2013, p. 152.

[45] Tony Veale and Guofu Li. "Specifying Viewpoint and Information Need with Affective Metaphors". In: ().

[46] R Weide. *The CMU pronunciation dictionary, release 0.6.* 1998.

[47] Geraint A Wiggins. "Searching for computational creativity". In: *New Generation Computing* 24.3 (2006), pp. 209–222.