Natural-Language Input for the Scone Knowledge-Base System

Yang Yang

December 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Scott E. Fahlman, Carnegie Mellon University, Computer Science Department and Language Technologies Institute
Lori Levin, Carnegie Mellon University, Language Technologies Institute

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Keywords: Natural Language Understanding, Scone, Construction Grammar

Abstract

Recent advances in AI have brought various NLP tasks to an extremely high level, for example in text classification, translation, semantics analysis, etc. However, we believe the ultimate goal of these NLP tasks is to perform a human-like understanding and then reason or analyze over the new learned knowledge, which should be a concept-level representation of the meaning of the utterance. Therefore, we decide to take a step forward and build a prototype natural language understanding system that could extract and store the knowledge from incoming texts. The whole system utilize Scone to store the extracted knowledge and Construction Grammar to build the interpretation of the text. Scone is a high-performance knowledge base system for storing symbolic representation of knowledge. It acts as a smart memory that could store implicit knowledge, support simple inference and flexible search, and reason over multiple context. Meanwhile, construction grammar studies the pairing of linguistic forms and underlying meanings, whose ”bottom-up” structure could effectively gets Scone-resident background knowledge into the game early and often. We first provide a well-defined and flexible interface to define new constructions along with an implementation of some most commonly used constructions. Then we introduced the Matcher and Constructor algorithms that is used to build new knowledge structure in Scone based on the input text and construction rules. Finally, we introduce the ”guess and go on” algorithm to support and manage different interpretations in a sequence of text input. Our current implementation is able to understand simple sentences and consequently, provide a new NLU front-end to Scone.

Acknowledgments

My advisor is cool.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Natural Language Processing (NLP) is a very unique part of Artificial Intelligence, not only because language is one of the most complicated and ambiguous thing in human society, but also because the development of NLP narrows the communication gap between human and computer. In recent years, with the development of deep neural net, NLP has gained huge improvement in various fields: text classification, translation, semantics analysis, etc. However, the ultimate goal of all NLP tasks is to perform a human-like understanding of some pieces of text and reason over the new knowledge extracted from the text.

We hold the belief that human language is, fundamentally, a tool for identifying a piece of meaning in my mind and building a good, effective approximation of that knowledge in your mind. And consequently, a real Natural Language Understanding (NLU) system should be able to go from incoming text, in context, all the way to a language-independent, concept-level representation of the meaning of the utterance,in other words ,the new knowledge it conveys. However, to build such system is a really complex task, where the system must include background knowledge, not contained in the utterance, to help with the understanding: resolving ambiguities, filling in missing-but-assumed details, and so on [7].

The goal of this project is to build a prototype NLU system that demonstrates the core features of this approach and to provide a foundation upon which future work in this area can build. There are two fundamental question in building such system: How to store knowledge and How to extract knowledge. In this project, I choose to use Scone for storing knowledge and Construction Grammar for getting the text semantics and the following two sections will briefly introduce Scone and Construction Grammar and the reason they are suitable for the NLU engine.

1.2 Scone Knowledge Base

Knowledge representation, which deals with representing real world knowledge in a format that could be used by computer, is also another wildly discussed problem in the field of Artificial In-

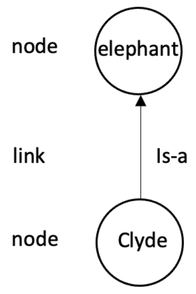


Figure 1.1: Scone Elements

telligence. For the purpose of my project, we want a knowledge based system that could support human-like thoughts and performance, specifically the symbolic part of memory. And Scone could be an ideal knowledge base to support human-like memory system.

Scone [3] [4] [5] is a high-performance, open-source Knowledge-Base System (KBS) that has been under development by the research group lead by Professor Scott E. Fahlman at Carnegie Mellon University since 2003. Scone, by itself, is a software component and active memory system and does not have a built-in decision-making component. It is designed to store the large network of knowledge and by using inheritance, it could store implicit knowledge that are not explicitly defined. Like other knowledge-base systems, Scone provides support for representing symbolic knowledge about the world: general common-sense knowledge, detailed knowledge about some specific application domain, or perhaps some of each. Scone also provides efficient support for simple inference: inheritance of properties in a type hierarchy, following chains of transitive relations, detection of type mismatches (e.g. trying to assert that a male person is someone's mother), and so on. Scone supports flexible search within the knowledge base. For example, we can ask Scone to return all individuals or types in the KB that exhibit some set of properties, or we can ask for a best-match if no exact match is available[4].

There are three features of Scone that are repeated used in this project:

- Clean separation of conceptual vs. lexical or linguistic information
  Scone may be viewed as a semantic network of nodes and links. The nodes in Scone represent concepts, not words or phrases. These nodes are generally written with curly braces, such as elephant, which represents the concept elephant and is tied to other concepts by relational links. We refer to Scone's nodes and links collectively as Scone elements. The names within the curly braces are arbitrary unique labels, but the concept elephant is tied to zero or more lexical items in English, and to additional lexical items in other languages. The concepts themselves are mostly universal and language independent – the concept elephant is more or less the same in every culture that knows about elephants – but Scone's multiple-context mechanism can be used to represent concepts that exist in one culture but not in another, or that are significantly different in some cultures.

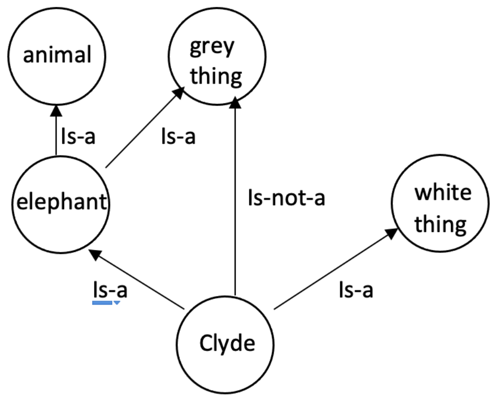


Figure 1.2: Scone nodes inheritance example (chain, multiple and exception)

- Frame-like representation with virtual-copy semantics
  In Scone and other frame-like KB systems, there are arbitrarily complex structured descriptions (prototypes) for each type represented: elephant, family, contest, and so on. These concept descriptions typically have many interrelated parts, attributes, and relations to other individuals and types. Upon creating a subtype or instance, all of the structure of the prototype description is efficiently inherited. In effect, the new subtype or instance becomes a virtual copy of the parent description; it can then be specialized by adding new information or by subtracting information that would otherwise be inherited.

- Multiple contexts in Scone
  Perhaps the most unusual aspect of Scone is its multiple context mechanism. A context is simply a node in Scone's knowledge base that represents a distinct world model within the larger Scone KB. There can be any number of these contexts within the Scone system. Every node N in Scone has a connection to some context-node C in which the entity represented by N exists. Every link L in Scone has a connection to a context node C within which the statement represented by L is valid. In Scone, contexts are hierarchical: a context C2 starts out as the clone of some existing parent context C1, inheriting all of C1's knowledge; then we can customize C2 by adding some additional knowledge or by explicitly subtracting some knowledge that would otherwise be inherited. This inheritance is handled efficiently by Scone's markerpassing algorithms. More details can be found in [4].

In a word, Scone is not a complete AI but rather a smart memory for symbolic knowledge used in AI or software applications. The open source of Scone, implemented in Common Lisp, could be found on Github and related information about Scone could be found in Professor Scott E. Fahlman's Knowledge Nuggets blog [5].

Then I would like to list a couple of Scone features that we think are essential to support human-like performance [6] and this is how Scone stands out comparing with other existing knowledge graph/base.

- The basic components of the Scone knowledge base are nodes and links, which together referred as elements. Nodes represent entities and links represent the relation between the connected nodes. Any element in Scone is the representation of a concept rather than a word or a phrase since intuitively, this is how human learns a piece of information where we learn the information about a certain conceptual group and such knowledge is independent of the word or the language describing it. Consequently, even in a single language (e.g. English), a node could match with multiple words (e.g. subway and metro might refer to the exact same transportation) and a word could match with multiple nodes (e.g. the word mouse could refer to animal mouse or computer mouse). There are some existing knowledge base that does not distinguish words with conceptual elements and we believe the way Scone manage and define its elements are closer to the way how human memory system works.
- The essence of Scone is a semantic network [13] and the backbone of such network is the is-a hierarchies of the nodes. Such hierarchy helps to maintain the inheritance of properties. Intuitively, this is very common in natural world, for example, the taxonomy in biology is a typical is-a hierarchy. A typical dog falls into Canis lupus species and therefore it will inherit the features of a wolf-like animal: four legs, a tail, carnivore, etc. Additionally, we believe supporting multiple inheritance is also essential to make a KBS closer to human-like memory. Scone elements could have multiple parents, for example, node {**Clyde**} could be a {**animal**}, {**elephant**}, {**smart thing**}, etc. at the same time. Finally, Scone also support exceptions in inheritance reasoning and it is consistent with common sense. When I say Clyde is an elephant, although Clyde is assumed to have four legs, it is acceptable that in later context, Clyde became a three-leg elephant.
- Although we still do not have a rigor reasoning of how human brain stores the information, it is generally accepted that our memory are capable of scaling up to a relatively large size. Furthermore, the memory system itself could also do some inference jobs very fast. For example, when we say ”Clyde is an elephant”, it would immediately came to our mind that Clyde has four legs and most probably Clyde is gray. We would like to have a KBS that is capable of doing such search and inferences in a relatively high speed even when the knowledge base grows to a large size. Scone differs from other KBS by using a maker-passing algorithm that could handle most kinds of search and inference in common-sense reasoning with a very high speed [3].
- Finally, we want to emphasize the importance of context in KBS. Consider the following example: People are able to do magic in Harry Potter. In general context, apparently human being in common sense cannot do magic. However, in the context of {**Harry Potter**}, human being are capable of doing magic. And the KBS should be able to store such different states in different worlds. The context wire of the Scone elements helps to support different but overlapping world models at once and support reasoning of episodic memory like before and after. These features helps Scone to perform a more human-like reasoning than other knowledge base systems.

Pattern	Meaning	Example
apple	apple	apple
X-ed	X :past	kicked
a/an X	a new individual X	an apple
X take Y for grated	X does not value Y	Clara takes me for granted
X's roommate	a person who is X's roommate	Clyde's roommate

Figure 1.3: Construction examples

1.3 Construction Grammar

Construction grammar (CxG) is a field of cognitive linguistics that studies the pair of the linguistic patterns and the underlying meaning. Constructions, including morpheme, word, idiom, passive, etc, is a very broad linguistic concept, where in fact any pattern could be recognized as a construction as long as some aspect or its meaning cannot be predicted from its component parts or other constructions [8]. While some linguistic theory treats lexicon and syntax separately, CxG consider all constructions to be part of a lexicon-syntax continuum [12].

There are a couple of reasons why we think CxG would be an ideal approach to a solid natural language understanding engine. CxG emphasize that any kind of information expresses through language could be conventionally associated with a particular linguistic form and therefore could be concluded from a certain construction. Thus, CxG could achieve a high level of full coverage and generalization across languages [10]. Another key reason that we favor CxG is that the way it works follows a "bottom-up" structure. Unlike context-free grammar, construction grammar starts from constructing small parts that eventually meets the construction of a larger part, while context-free grammar needs to start from the whole text and recursively break up into small parts. Considering a NLU engine, such "bottom-up" structure gives the input text more freedom and also helps the system to deal with incomplete or unstructured text.

Additionally, Construction Grammar is great to combine with Scone in the following aspects:

- Construction rules try to find "meaningful fragments" of language in an incoming stream of text. In construction pattern, these fragments are generally restricted to the type of entity that can fit into that part of pattern – for example, the pattern can restrict one certain fragment to be a person. Scone has powerful, built-in machinery to answer questions like "Is X a Y?" or "Can X be a Y?". In this case, person type is directly mapped to {person} element in Scone and by asking Scone "Is X a {person}?", the system could tell if the text could fit to that part in the pattern. This gets Scone-resident background knowledge into the game early and often and simplifies the process to match text with construction pattern.

- If a construction pattern is matched (tentatively), is it associated with a rule that gives instructions for building a piece of knowledge structure in the Scone knowledge base. Scone can quickly determine whether this new knowledge is consistent with what it al-

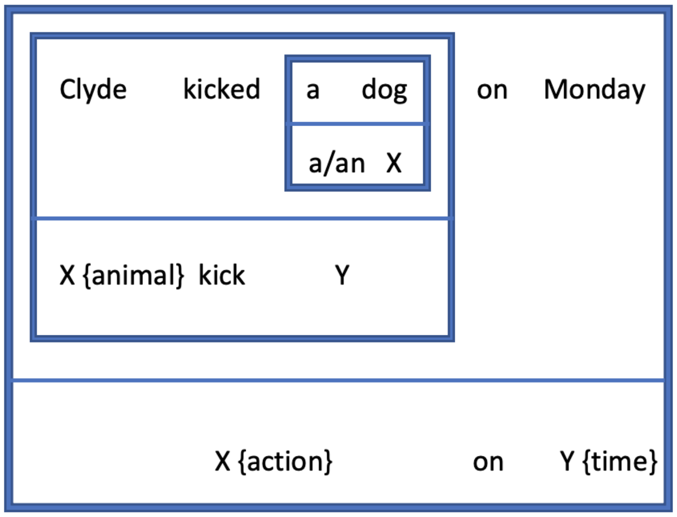


Figure 1.4: Example of construction grammar

ready knows – if not, it should look for a different interpretation. The ”bottom-up” structure of construction grammar also speeds up the whole process since the system start by checking small pieces of texts.

- If there are several possible matches for some incoming piece of language, the multiple context feature of Scone can build a knowledge structure corresponding to each of these possible interpretations, each in a different Scone context. If one interpretation is later ruled out, we can simply discard that context and all its contents, and treat the other one as likely being the intended meaning. For example, ”The mouse was on my desk, next to the computer keyboard...” will create two interpretation context in Scone, one with computer mouse and one with animal mouse. The later text ”... eating some cheese and stroking its fur.” will eliminate the context with computer mouse and move on with animal mouse.

- The context mechanism, which creates distinct world models within that larger Scone KB, can also be used to represent information that changes from one time-context to another, perhaps as a result of some action or event. It can also represent modal constructions, such as ”I want to stay at home tonight, but I should go to the meeting.” The two possible worlds are represented in two distinct world model that share most of their information. Scone can the reason about what is true in one of the other – maybe needing transportation if going to the meeting.

CxG itself is a family of theories and there are a couple of existing formalized Construction grammar frameworks that are mature and have a solid implementation: Embodied Construction Grammar[1], Fluid construction grammar[11], etc. Details of how constructions are defined and generalized in this project will be discussed in the next Chapter2. The focus of this research is not to build a solid construction grammar framework, but to build an NLU-front-end to Scone and provide an interface for linguistics to experiment with self-defined construction rules at the same time.

1.4 Thesis Structure

The remainder of the thesis is structured as follows. In Chapter 2, we will discuss how the constructions are saved the in system and how user could define new constructions. In Chapter 3, we will show the main algorithms of the core NLU engine. In Chapter 4, we will implement an example CxG for the purpose of parsing simple texts and demonstrate the output of the system. And finally, we will give the conclusion of the research in Chapter 5 together with some possible future directions.

Chapter 2

Constructions

2.1 Overview

Under the setting of CxG, grammatical construction refers to the paring of linguistic form and content. Therefore, to represent a construction, there are three things we need to figure out: how to represent the form of a piece of text, how to represent the semantics and how to set up the connection between forms and semantics. To make things easier, I decide to represent text as a sequence of variables and strings, where the variables serve as the bridge to connect form and semantics. Specifically, the constraints on the variables help to define the form of the text and the semantics is a function that takes in the variables and outputs a Scone element representing the meaning.

Figure 2.1, 2.2, 2.3 demonstrates how the three constructions in Figure 1.4 could be defined using variables to connect the pattern and semantics. The New Individual Construction can match with noun phrases like ”a dog”, where the the $?x$ variable match with the Scone type node {dog}. The semantics of this construction will return a new individual node {dog 0-2702}, which match with the $?y$ variable in Transitive Action Kick Construction since {dog 0-2702} is a {physical object}. Then ”Clyde kicked a dog” will match with the Transitive Action Kick Construction and generates a new action {kick 0-2722}, whose {action agent} is {Clyde} and {action object} is {dog 0-2702}. Finally, the new individual kick action {kick 0-2722} matches the ?x variable in Action with Time Preposition Construction, and change the context of {kick 0-2722} to {Monday}.

Noun Phrase New Individual Construction	
Pattern	(a, an) ?x
Variable Constraints	(?x :noun :type)
Return Tag	:noun
Semantics	Scone create new individual of type ?x

Figure 2.1: Example new individual construction

Transitive Action Kick Construction	
Pattern	?x (kick, kicks) ?y
Variable Constraints	(?x {animal} :noun) (?y {physical object} :noun)
Return Tag	:verb
Semantics	Create new individual of {kick} : ?v' Set ?x as the {action agent} of ?v' Set ?y as the {action object} of ?v'

Figure 2.2: Example transitive action kick construction

Action with Time Preposition Construction	
Pattern	?x (in at on) ?y
Return Tag	:verb
Variable Constraints	(?x {action}) (?y {time reference})
Semantics	set the context element of ?x to be a ?y

Figure 2.3: Example transitive action kick construction

In order to formalize the construction and make it easier to compact with the Scone knowledge system, we need to specify the following things while defining a new construction:

- Variables and Corresponding Constraints
  First, we need to specify the variables to use in patterns and semantics and for each variable, there could be a list of constraints. As demonstrated in the example, a constraint could be a Scone element, a string, or a tag. Details of how the constraints are used will be discussed in 2.2.2.

- Pattern
  The Pattern of a construction is used to match with the form of some raw text. The pattern is formed by a sequence of entities, where each entity could be either a list of strings or a variable. A piece of text is matched with a string list entity if and only if the text is exactly the same as one of the strings in the list. A piece of text is matched with a variable if and only the Scone element (meaning) of the text meets the constraints of the variable. Details of the matcher algorithm will be discussed in 3.3.

- Return Tag
  We also give a return tag to each construction in order to make the whole CxG engine compatible with Scone knowledge base. In Scone, the mapping of a string to an element always

comes with an associated syntax tag. Since when we try to map a variable to an element, it could be a direct mapping or a construction match, therefore, giving the construction a return tag makes the form consistent. Possible tags could be: :noun, :adj, :verb , :relation, etc.

- Modifier
  Modifier is used when we want to affect a certain variable before matching the text with a Scone element. Details will be discussed in 2.3.

- Action
  The action of the construction represents the semantics of the construction. The action part should be a Lisp program that could put new knowledge into Scone. Previously specified variables can be used in the program and the output of the program should be a Scone element or a list of Scone elements that correspond with the return tag. In the CxG system, the action will be treated as a lambda function that takes in the specified variables. Details of how the system applies the action will be discussed in 3.4.

- Parent
  The parent of a construction is either another construction or nil. Specifying the parent helps to build the construction hierarchy, which will be further discussed in 2.5.

2.2 Variables

As mentioned, variables serve as the bridge to connect the form and the semantics of text. The constraints of the variable help to define the form of text and the value it takes helps to define the semantics. Therefore, we need to carefully manage both sections.

2.2.1 Value

Since we are matching raw text with variables, the variable value should try to contain as much information about the text as possible and we choose to store the morphological information and semantics of the text. Morphology studies the form of words while semantics include the underlying meaning of the text. Both fields give distinct but essential information to use in the construction.

In general, semantics are represented as Scone elements and morphological information are reflected as a list of keywords. For example, when we try to match the variable with ”kicks”, the corresponding variable value should be ({kick} :singular). When we try to match the variable with ”was”, the corresponding variable value should be ({are} :past :singular). When we try to match the variable with ”his”, the corresponding variable value should be (X :possessive), where X is whatever ”he” is referring to. Details of how the system gets the semantics and morphological information will be discussed in the matcher section 3.3.

2.2.2 Constraints

One variable can have multiple constraints while each constraint can be a string, a Scone element or a keyword. String constraints aim to constrain directly on the text, while Scone elements aim to constrain the meaning of the text. Keyword is the most flexible constraint as long as it is well defined in the system. I will elaborate more on how to use each type of constraint.

- String Constraints

  In a lot of cases, we want the text to have an exact match with a word or a phrase. For example in the above kick construction, we want the verb to be exactly "kick". Then we can just "kick" string to constrain the variable. Moreover, different tenses of "kick" should also match the construction. Consequently, we again choose to use the morphology parser and if the text wants to match the constraint, the root text needs to exactly match the string constraint. If we have multiple string constraints, it is following an "or" logic that the text needs to match one of the string constraints.

  For example, we have a variable with constraints: "kick", "hit", "eat". Then "kicks", "kicked", "hit", "ate" should all match with the variable constraints.

- Scone element Constraints

  One of the major advantages of combining CxG with Scone is that the implied knowledge of the text could be represented. For example, when we say "A kicks B", we are actually assuming A has legs and in most cases A is an animal. Therefore, in the kick construction rules, we would use the Scone element {animal} to constraint the action agent. To check if a text meets the constraints, the system will first get the Scone element value of the text (as discussed in the 2.2.1), then call the Scone built-in **simple-is-x-a-y?** function.

  For example, we have a variable with constraint {animal}, "elephant" should match with the variable since "elephant" will be parsed to {elephant} and apparently {elephant} is a {animal} (In Scone, the two nodes will have a is-a link).

- Keyword Constraints

  Additionally, one might want to constrain the variable on other aspects, for example syntax tag, whether the text represents a list of elements, etc. I've implemented a couple of keywords that I find the most useful:

  - Syntax tag (:noun, :verb, :adj, :relation, etc.)
    Note when we try to get the variable value of a piece of text, it goes through either Direct Scone mapping or the CxG engine, where both ways come with a syntax tag.

    For example, if we have a variable with :noun tag and try to match with "elephant",

”elephant” will be mapped to ({elephant}, :noun) and the syntax tag matches.

- Individual or Type (:indv, :type)
  In Scone, there are two different type of nodes: individual nodes and type nodes. Such two type of nodes tries to catch whether the natural language is referring to a general type or an individual. For example, when we say ”Elephants are smart”, we are actually talking about the entire elephant species, which is the type node. However, when we say ”I saw an elephant”, we are referring to a specific elephant, which is the individual node. Distinguishing these two kinds of nodes is essential in constructions.

  For example, in the above-mentioned new individual construction, the ?x variable has to be a type node. Say we have a new elephant individual {Clyde} and it wouldn’t make any sense if we say ”a Clyde”.

- List (:list)
  Consider the following two sentences: ”Wesley and Yang are tall”, ”Wesley and Yang are friends”. Although both sentences have a similar structure and exactly the same subject ”Wesley and Yang”, we should treat them differently when considering the semantics. ”Wesley and Yang are tall” means exactly the same as ”Wesley is tall” and ”Yang is tall”, so ”Wesley” and ”Yang” are just sharing the structure of the sentence but does not affect each other in the semantics. However, to understand ”Wesley and Yang are friends”, the system needs to create a new relation between {Yang} and {Wesley}. Therefore, when dealing with the first sentence, the subject variable does not need both {Yang} and {Wesley}, but rather apply the action on both of them separately, while dealing with the second sentence, the subject variable should include both {Yang} and {Wesley}. Therefore, we need to use :list as a keyword constraint to tell the system whether we need a list or a single element.

  Continuing on the above example, if we have a variable with :list constraint and tries to match ”Wesley and Yang”, the action of the construction will be applied once with ’({Yang} {Wesley}) as the variable value. If we have a variable without :list constraint and try to match ”Wesley and Yang”, the action of the construction will be applied twice with {Yang} and {Wesley} as the variable value separately. Usage of :list constraint will be demonstrated in Section 4.5.

One last thing that needs to be settled is the logic between multiple constraints. Currently, only string constraints follow ”or” logic, and all other remaining constraints will follow ”and” logic. That is to say, the system will first check if the text meets one of the string constraints and then check if it meets all of the rest constraints. Ideally, we would want the logic to be as flexible as possible so that the user could constraint the variable with more flexibility. This could be one of the future works of the project.

Person Belief Construction	
Pattern	?x (”believes that” ”believe that”) ?y
Variable Constraints	(?x {person} :noun) (?y :relation)
Modifier	(?y (new-indv nil {person}))
Return Tag	:relation
Semantics	Create an eq link between ?x and the context wire of ?y return ?y

Figure 2.4: Example person belief construction

2.3 Modifier

Since the construction grammar follows a bottom-up structure, it is possible that upper construction wants to modify the bottom constructions. However, the bottom constructions were already settled down and the action cannot be directly modified. Therefore, we need to use modifier to trigger an action before acquiring the variable value.

Let’s take a look at the sentence ”Yang believes that Clyde is an elephant”. Assume the CxG engine is already well built and the text ”Clyde is an elephant” will be matched to a new is-a link {Is-A 0-2780}. In order to represent the actual semantics of the original sentence, we need to change the context of the is-a link to {Yang} since it is Yang’s belief. Consequently, we need to use modifier to define the Person Belief Construction 2.4. Note that using the modifier, the system could create the Is-A link under a new person context and the action of the construction will create an equal link between the new person and {Yang} so as to correctly represent the original semantics.

Modifiers are also commonly used for prepositional phrases, where we will discuss its usage in Section 4.6.

Currently, the system only supports context as modifiers since we see it the most useful when defining constructions. But any type of modifiers could be added if necessary, like tense or uncertainty keywords etc.

2.4 How Constructions Are Stored

Since in the next chapter, we will dive deeper into the algorithm of the NLU system, we would like to go over how we store the construction from a code level. We create a new class in Lisp with five slots: ret-tag, pattern, variable-constraint, modifier and action. And I will use the construction in Figure 2.4 for the purpose of demonstration.

ret-tag: Simply store the return tag of the construction, :relation in this case.

pattern: Pattern is saved as a list but the trouble is how to store the variables. We choose to use numbers, which in this case it would be (0, ("beleieves that", "believe that"), 1).

variable-constraint: The constraints are also saved as a list where the index of the list is the corresponding variable, and each element of the list is a list of single constraints. In this case, it would be (({person}, :noun), (:relation)).

modifier: Modifier is dealt similarly as variable-constraints where in this case (nil, (new-indv nil {person})).

semantics: Semantics are saved as a lambda function, where in this case, (lambda (?x ?y) (...)). Therefore, when we get the value of the variable, we could directly apply the lambda function to build knowledge structures in Scone.

2.5 Hierarchy

If we consider the form of text as a set of texts, it is common to find that a subset of texts usually inherits semantics from the superset. For example, "Wesley kissed the dog" and "Wesley killed the dog" definitely have different semantics but both of them match with a general transitive action construction so we know Wesley is the agent of the action and the dog is the object of the action. Therefore, it is natural to build a hierarchy structure for the constructions. To make things easier, we assume each construction could only inherit from one parent. And a natural way to represent such structure is a tree. If a construction does not have a parent, it will become the root of a new tree. And all constructions forms a forest 2.7.

A very intuitive usage of this hierarchical structure is tense. The General Transitive Action Construction 2.5 only deals with the morphology tags of ?v and determines the context. The Transitive Action Kick Construction 2.6 can inherit General Transitive Action Construction since the text form represented by kick construction is a subset of the form represented by general action construction. Such inheritance will save from repeated tense actions as the system will first execute the action from parent constructions. Detailed algorithms of how the CxG utilize such hierarchy will be discussed in 3.4.

General Transitive Action Construction	
Pattern	?x ?v ?y
Variable Constraints	(?x {thing} :noun) (?v :verb) (?y {thing} :noun)

Semantics	If ?v has :past keyword, in-context {past}
	If ?v has :future keyword, in-context {future}

Figure 2.5: Example transitive action (tense) construction

Transitive Action Kick Construction	
Pattern	?x ?v ?y
Variable Constraints	(?x {animal} :noun) (?v kick :verb) (?y {physical object} :noun)
Parent	General Transitive Action Construction
Semantics	Create new individual of ?v : ?v' Set ?x as the {action agent} of ?v' Set ?y as the {action object} of ?v'

Figure 2.6: Example transitive action kick construction

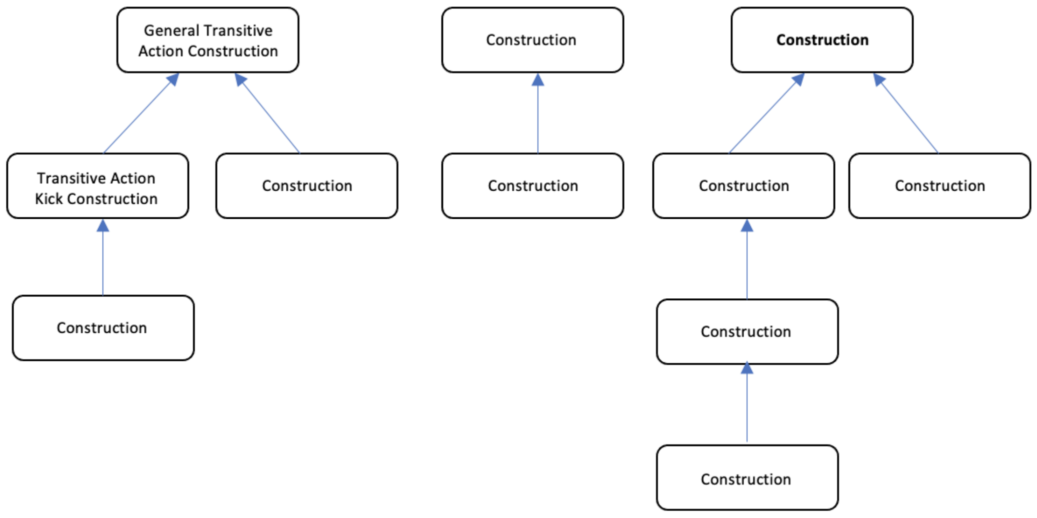


Figure 2.7: Constructions hierarchy (forest)

Chapter 3

NLU Engine

3.1 Overview

Chapter 2 talked about how we formalize constructions and at the same time, provide a user-friendly Lisp interface to define new construction rules. While the main purpose of this project is to build a real natural language understanding system based on the constructions defined, and this chapter will be showing the details and algorithms of how we construct such engine.

Figure 3.1 demonstrates the general workflow of the NLU engine. The system is able to take in a sequence of text and for each input, it will go through the steps showed in the workflow and build the corresponding knowledge structure in Scone. The engine consists of five major components: Constructions, Context, Matcher, Constructor, Text and Context Memory, and the Text Reader.

- Constructions is the construction forest we talked about in 2.5. For each input text, the system will check every construction, according to its hierarchy.
- Context is one of the most important things in doing natural language tasks. Texts could have different meanings in different contexts. Therefore, we need to carefully represent the context. Our Context consists of two parts: general language context, using the Scone builtin mechanism, and a referral context, which is used to save previously referred objects. Details will be discussed in 3.2.
- Matcher, as the name suggests, is used to match text with constructions. The Matcher takes in raw text and a construction, and returns the variable values if it matches. Detailed algorithm will be shown in 3.3.
- Constructor serves as a central unit in processing the most recently input text. It collects the output of the Matcher and parses the final construction results to the Core Engine. Details will be described in 3.4.
- Core Engine consists of a Text Reader and a Memory. The text reader is the final layer of the system that manages the result and returns it to the user. It is also responsible for updating the memory which consists of previously unused results (text and context). The memory helps the whole system to do backtracking and will be discussed in 3.5.

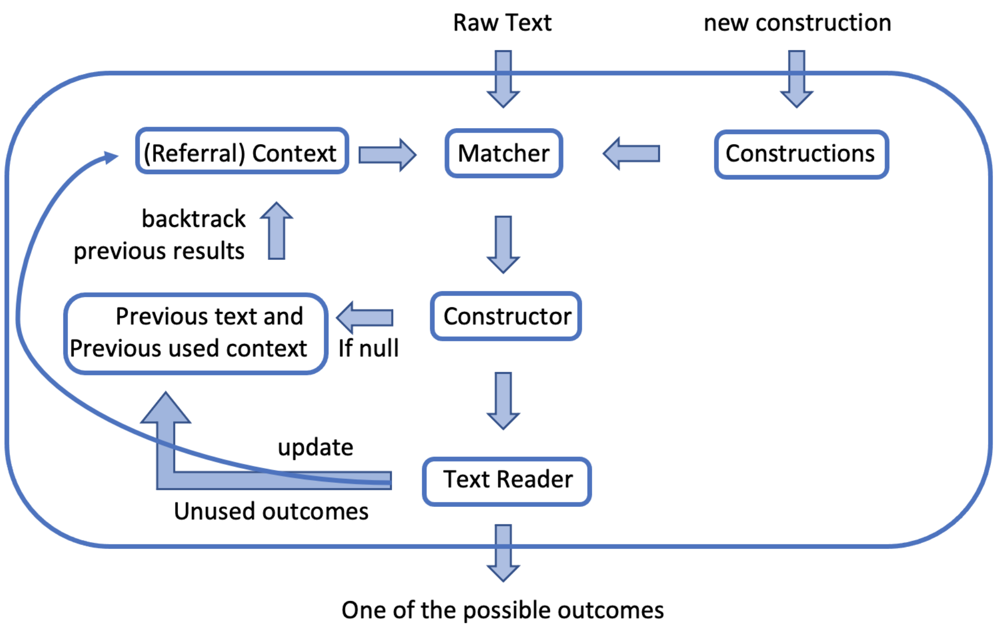


Figure 3.1: NLU engine workflow

The five components work together to form the NLU engine. The user can input a sequence of texts into the engine, where for each text, the matcher will try to match the text with the constructions saved in the system. Then, the constructor will take all the results from the matcher and construct the semantics of the specific text. If there's no valid semantics in the current context, the constructor will backtrack to the previous unused result and reconstruct the current text. If there is at least one valid semantics, the text reader will take and return one of the possible results and store the unused results.

The most important feature of the entire engine is guess and go on. Since words, phrases, sentences all could have multiple meanings, and we want to be as inclusive as possible, so what the system does is choose one meaning and move on but still keep the possibilities of others. Details of the algorithm will be discussed in Section 3.5.

3.2 Context

To make an AI system able to understand natural language, the first thing we need to figure out is how to represent the context. Natural language might have different meanings under different contexts and context also represents some underlying assumptions in the text. Therefore, in our NLU engine, we decide to use two components to represent context: Scone context node and referral context.

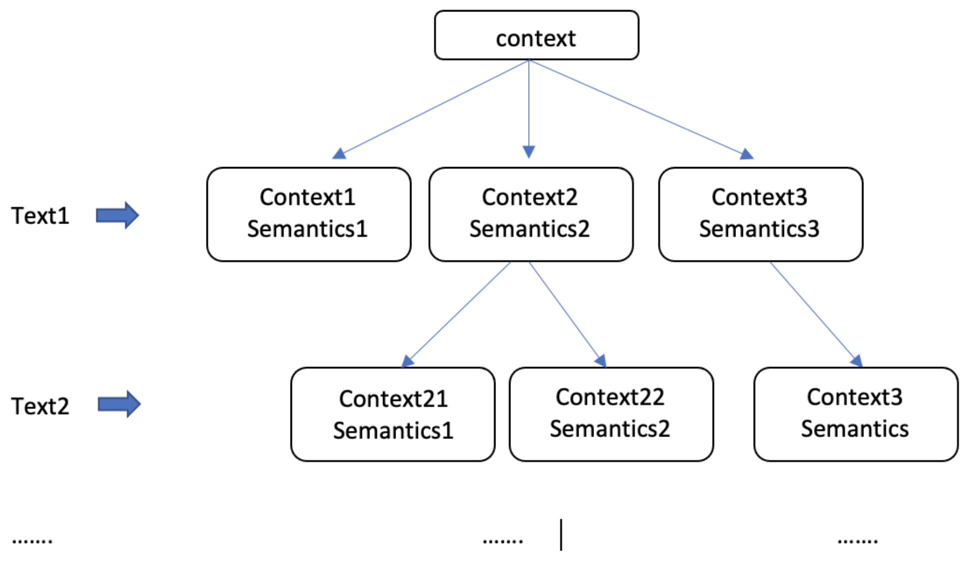


Figure 3.2: Scone context for semantic path management

3.2.1 Scone Context

Scone has a built-in Context mechanism where every Scone element has a context-wire. Context is simply a Scone node and it can be used to tie a node to the context in which it exists, and used to tie a link to the context in which it is true. Consequently, this gives us two types of usage.

A very natural way is to use Scone context is to represent the context of the sentence. For example, considering the three sentences ”Clyde is an elephant”, ”Clyde was an elephant”, and ”Yang says ’Clyde is an elephant’”, they all represent the same is-a link between Clyde and a new individual elephant. However, every sentence has a different context. ”Clyde is an elephant” represents the is-a link under a {general} context; ”Clyde was an elephant” represents the is-a link under a {past} context; and ”Yang says ’Clyde is an elephant’” represents the is-a link under the context {Yang}.

Additionally, Scone Context can be used in the system to manage different possible interpretations (multiple outputs of the constructor). As we know that every Scone node has a context wire, we could create dummy children of the current context to manipulate different paths so that the actions in different paths do not affect each other. For example, in {general} context, an input ”Clyde is a mouse” has an ambiguous meaning that Clyde can be an animal mouse or a computer mouse. If we are not using multiple context, when we put different semantics into Scone, {Clyde} would become an animal mouse and a computer mouse at the same time. Therefore, what the system should do is to first, create two dummy context {general 0-4164} and {general 0-4165}, then proceed Clyde is an animal mouse in {general 0-4164} and proceed Clyde is a computer mouse in {general 0-4165}. Basically, whenever the text has multiple meanings, the system will create children of the current *context* and proceed each path under one of the child contexts as demonstrated in Figure 3.2.

3.2.2 Referral Context

Another important factor of context is that it saves previously mentioned information which gives some assumptions in the current text. For example when we say ”the elephant”, that means there was at least one individual elephant mentioned in the previous text and ”the elephant” is referring to that specific elephant. To save these previously mentioned individuals, the system uses referral context.

The referral context is saved as a stack in the system. Additionally, when a new node is pushed into the referral, the system will first check if the node is already existed in the referral and remove it, and then push it to the top of the stack. The intuition of using a stack is that when people refer to an individual in a previous context, they often refer to the most recently mentioned ones.

For example, starting from a NIL referral context, if the system first creates an {elephant} individual name {Clyde}, the referral context will be ‘({Clyde}). Then if the system creates a new {elephant} individual {elephant 0-3141}, the referral context will be ‘({elephant 0-3141} {Clyde}). Finally, if the system gets {Clyde} again, the referral context will put {Clyde} to the front and become ‘({Clyde} {elephant 0-3141}). Now, if the system gets an input ”the elephant”, it knows it’s referring to {Clyde}.

3.3 Matcher

Before we apply the construction actions, we need to know if a piece of raw text could match with the specified form of the construction and that’s the role of the Matcher in the system. We will first discuss morphology, and then go over the main algorithm of the Matcher. Additionally, details of how the system deals with some special cases like names and pronouns will be shown. Finally, we will give some ideas in partial matching.

3.3.1 Morphology

A critical part of the Matcher is to determine if a piece of text can be fit into a variable in the pattern. As we discussed in section 2.2.1, the value of a variable consists of a Scone element and a list of morphology keywords. From the figure 3.3, we can see before checking the constraints and getting the value of the variable, the variable will first be parsed into a morphology engine and get the morphology tags.

Morphology is a very broadly researched area in linguistics and NLP. A naive idea of a morpheme-based morphology parser is to transform any word into its root word and a list of keywords that the morpheme implies. For example, taking in ”kicks” should return ”kick” and :singular. Additionally, I want to extend the idea of morphology to not only words but also phrases. For example, taking in ”will kick” should return ”kick” and :future. Since this is not the focus of my research, current implementation of the morphology engine simply checks the suffix to determine the tense of verb and whether the a noun is possessive. User can replace the morphology

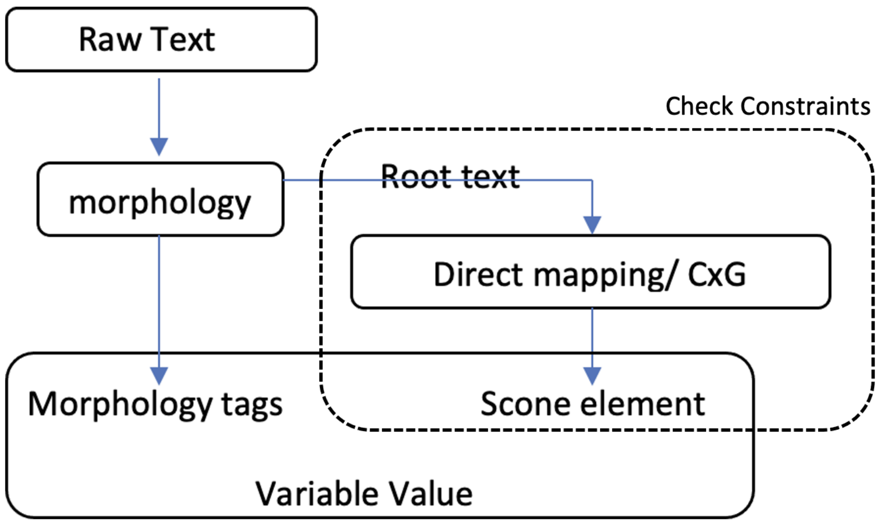


Figure 3.3: Flow graph for variable match

parser with more mature systems as long as the format match.

3.3.2 Single Entity Matcher

The idea of the Matcher is to repeatedly check if a piece of the original text matches with a part of the pattern. Therefore, we need to first figure out how to determine if a text matches a single entity in the pattern, where the algorithm is shown in 1. Based on the type of the pattern component, we can split into the following two cases:

- If the pattern component is a variable (represented as integers in the system), it will following the flow showed in 3.3: first get the morphology tags and root text, and then use variable match to get the corresponding Scone context and check if it meets the variable constraints along the way.

  For example, say the text T is ”kicks”, pattern P is 0 and constraint list Cl is ((”kick” ”hit” :verb) ...). The system will first use the morphology engine and get (”kick” :singular). The root text ”kick” and the constraints (”kick” ”hit” :verb) will be parsed to variable match. The root text matches one of the string constraints (”kick” and ”hit”). Then, the system will call LOOKUP-DEFINITIONS(”kick” :verb), which is a built in function in Scone, and gets {kick}. Finally, {kick} and the morphology tag :singular will be concatenated and saved as the variable value for variable 0.

  Say the text T is ”an elephant”, pattern P is 0 and constraint list Cl is ((animal :noun) ...). No morphology keywords will be obtained and the root text ”an elephant” and the constraints (animal :noun) will be parsed to variable match. Then, the system will call

CONSTRUCTOR(”an elephant” :noun), which will be discussed later in the next section, and gets a new individual in Scone say {elephant 0-2702}. Since {elephant 0-2702} is a {animal} in Scone, the result meets the constraints and the final variable value will be ({elephant 0-2702}).

- If the pattern is a list of strings, the function simply needs to check if the input text has an exact match with one of the strings in the list. For example say the input text T is ”in”, pattern P is (”in” ”on” ”at”), then apparently, the text matches the pattern and ”in” will be returned. If T is ”in it”, then there’s not an exact match and the function will return nil.

Algorithm 1 Single entity match

```
$T$ is the root text
$C$ is the constraints for this specific variable
procedure VARIABLE MATCH($T$ $C$)
    Check if $T$ match with the string constraints in $C$
    Extract $st$ from $C$                                    ▷ $st$ refers to the syntax tag constraints
    for $e$ in LOOKUP-DEFINITIONS($T$ $st$) + CONSTRUCTOR($T$ $st$) + special cases do
        if $e$ meets type constraints in $C$ then
            collect ($e$, context)
        end if
    end for
end procedure

$P$ is a single component of the construction pattern
$Cl$ is the entire variable constraints of the construction
procedure ONE ELEMENT MATCH($T$ $P$ $Cl$)
    if type of $P$ is integer then
        $RT, Morph \leftarrow$ calling morphology engine on $T$          ▷ $RT$ is the root text
        return (VARIABLE MATCH($RT$ $Cl[P]$), $Morph$)
    else if find $T$ in $P$ then
        return $T$
    else
        return NIL
    end if
end procedure
```

3.3.3 Main Algorithm

The responsibility of the Matcher is to determine if the text matches with the construction pattern. Note that as we discussed in Chapter 2 2, pattern is actually a list of entities (strings or a variable). Therefore, we need to split the text into several pieces so that each one could match the pattern component correspondingly. What we choose to do is to use a tokenizer to split the raw text into tokens. It would be very rare that a single word is split into two components in the pattern, therefore, it is safe to assume that every component of the pattern should match a sublist of the token list. In the current implementation, the system is just using a naive tokenizer to split the text into a list of words and punctuation (comma, etc), and users can replace it with other more mature tokenizers.

Consequently, the objective of the main algorithm is to take in a token list and a construction and return all possible variable value combinations.

Algorithm 2 Matcher

```
  $Tl$ is the token list get from the tokenizer
  $Pl$ is the construction pattern
  $Cl$ is the construction variable constraints
  procedure MATCHER($Tl$ $Pl$ $Cl$)
      if only one of $Tl$ and $Pl$ is null then          ▷ text does not match with construction
          return NIL
      else if both $Tl$ and $Pl$ are null then                                    ▷ base case
          return (nil, nil, ..., nil, current context)     ▷ number of nil is the number of variables
      else
          for $i$ from 1 to len($Tl$) - len($Pl$) +1 do
              $T \leftarrow$ join-tokens $Tl$[:$i$]
              $r \leftarrow$ ONE ELEMENT MATCH($T$ $P$[0] $Cl$)
              if type of $r$ is String then
                  collect MATCHER($Tl$[$i$ :] $Pl$[1 :] $Cl$)
              else
                  for $e$, $ctx$ in $r$ do
                      set context to $ctx$
                      for $rr$ in MATCHER($Tl$[$i$ :] $Pl$[1 :] $Cl$) do $rr$[$Pl$[0]] = $e$ collect $rr$
                      end for
                  end for
              end if
          end for
      end if
  end procedure
```

Based on the single entity match function defined above, we can construct the Matcher algorithm as shown in Algorithm 2. Note when we try to match a pattern with a token list, the first component of the pattern should always be matched with a "prefix list"(sublist starting from index 0) of the token list. Consequently, the function will first loop through all possible ”prefix list” and apply one element match with the first component of the pattern. For every match we get, the function will update the context and recursively call the Matcher on the rest of the token list and pattern list. Finally, the Matcher will return all possible matched results where each result is a list of variable values and the context that the values exist.

Note in the Matcher algorithm, the components are matched sequentially from left to right and the context is also updated accordingly. The reason we choose to do matching in this way is that we want an AI system that could imitate human behaviors especially in the task of natural language understanding. Human normally read the text from left to right despite of its syntactic structure. After a part of the text (prefix list of the token list) is read, humans will naturally process the text (match the semantics and update context) and keep on reading (recursively call the Matcher).

Now I'll give some examples to show the behavior of the Matcher algorithm.

For example, we have an input text "Clyde's friend" and construction ((?x :noun) (?y :noun :type)). The tokenizer will first break the text into ("Clyde's" "friend"). Then the system will try to do one-entity-match with "Clyde's" and (?x :noun), which will return ({Clyde} :possessive). Now, according to the Matcher algorithm, the system will recursively call Matcher on "friend" and (?y :noun :type), which will return ({friend}). Finally, the Matcher on "Clyde's friend" and ((?x :noun) (?y :noun :type)) will return ((({Clyde} :possessive) ({friend}) some-context)).

If we are calling Matcher on the text "Clyde is a mouse" and construction ((?x :noun) ("is a" "is an") (?y :noun :type)), similarly, the tokenizer will first break the text into ("Clyde" "is" "a" "mouse"). Then the system will try to do one-entity-match with "Clyde" and (?x :noun), which will return ({Clyde}). It will also try one-entity-match on "Clyde is" and (?x :noun) but these two does not match. Now, the system will recursively call Matcher on "is a mouse" and (("is a" "is an") (?y :noun :type)). It will first try one-entity-match with "is" and ("is a" "is an") but does not match. Then "is a" will match with ("is a" "is an") and go with calling Matcher on "mouse" and ((?y :noun :type)). Note, since mouse has two interpretations, one-element-match will return both ({mouse} context1) and ({computer mouse} context2). Finally, combined, the output of Matcher on "Clyde is a mouse" and ((?x :noun) ("is a" "is an") (?y :noun :type)) would be ((({Clyde}) ({mouse}) context1) (({Clyde}) ({computer mouse}) context2)).

3.3.4 Special Cases: Names and Pronouns

In the variable match function (Algorithm 1), the system checks the text through Scone direct mapping and Constructions. However, there are some cases that the text itself has ambiguous or various meanings and it's hard to formalize in the way we define constructions. To demonstrate how I deal with the special cases, I will elaborate on two very commonly used cases: names and pronouns.

Names are complicated not only because people could give a name to almost everything but also sometimes they don't even mention what the name is referring to. For example, when we say "Tony and Wesley are friends.", we do not need to specify Tony is a male in advance. However, in another context, Tony can be an elephant, which means the system cannot just simply assume Tony is a male person. Therefore, we need a more complicated mechanism to process Names.

First, we need to guess if the text might be a name. During the variable match, we say the text might be a name if we see the text and constraints meet the following conditions:

1. Every first letter of the words are capitalized
2. Only :noun can be the syntax tag in the constraints
3. The constraints can only constrain the variable to be an individual node
4. The Scone element constraints (the parents of the variable) cannot be an {intangible} instance

Then the system creates a new Scone element with the text as the English name. For every Scone element constraint, the system also uses a new is-a link to connect the new node with the constraint node. For example, if the Matcher takes in "Clyde hates mouses" and the construction (?x {animal} :noun)(?v "hate")(?z {thing} :noun), Clyde, as a name, should match with ?x. So the system first create a new {Clyde} node, then assert {Clyde} is an {animal}.

Together with Names, pronouns are also not trivial. Consider the following text: "Tony and Ellis went to the park yesterday. He bought an ice-cream in the park.". Who bought the ice-cream? To commonsense, Tony is a male's name, so we would guess Tony bought the ice-cream. What if the following text says "Tony is a little girl." Now, who bought the ice-cream? Probably it was Ellis. From the example, we can see that what's clear about pronouns is that it's referring to a previously mentioned agent, but what's unclear is who/what exactly it's referring to. Fortunately, Scone provides powerful, efficient support for determining whether a known entity X is or can be a member of known type Y, which could help the system to screen out the most probable results. Then, similar to what the engine has been doing, the system keeps all the possibilities and tries all possible meanings. To be more specific, here's how it deals with the most commonly used pronouns:

- he, him
  Since "he" normally refers to a previously referred male, the system looks into the referral context and gets the elements that are male person. However, in English, people normally don't specify the gender of a person and it is also hard to refer to the gender directly from the name. Therefore, the system will use the Scone method is-x-a-y?. When the element acquired from the referral context can be (:Maybe or :Yes) a male person, the system will take the element, create a new is-a link under a new context, assign the element to "he" and move on.
- she, her
  This is like "he" and "him" except that the pronoun refers to a female person.
- it
  This is also like "he" and "him" except we need the element to be not a person, so the system only select elements that get :Maybe or :No from the is-x-a-y? method and create a new is-not-a link under a new context.
- they, them
  "they" normally refers to a list of or a set of objects. Therefore, the system will just look into the referral context and acquire any list object or type nodes in the referral context.

Sometimes, “they” and “them” can also refer to a single individual of unknown gender, for example “If someone hits me, I will hit them back.” We haven’t implemented this case yet but it should be something to take into consideration in the future work.

3.3.5 Partial Match

Partial matching is exciting since, in the real world, texts are often not well formalized, especially in dialogues. In the process of matching text with patterns, if the system has a well-built partial matching mechanism, the text could be much less restricted. However, this is a pretty broad topic and could go really deep (a probability-based algorithm, etc.). Now, the system only supports a pretty naive partial matching just to show the potential of a CxG based NLU engine.

The intuition is really straightforward: we have two types of components in a pattern: string list and variables. While variables play a role in the semantics, string patterns are more of a structural usage and do not affect the meaning. Therefore, for our native partial matching, the text is allowed to miss one string component in the pattern (the pattern length needs to be at least 3 so that not a large portion is missing).

For example, in the figure 2.3, we showed an Action with Time Preposition Construction with pattern (?x {action}) (in at on) (?y {time reference}). The component (in at on) is just the prepositions to support the whole structure but does not have any direct meaning in the construction semantics. As demonstrated, ”Clyde kicked a dog on Monday” is a perfect match with the construction as ”Clyde kicked a dog” constructs a new action, and {Monday} is a time reference. Consequently, in our current system, ”Clyde kicked a dog Monday” can partially match with the construction since it still has the essential variable components.

Another example that such partial matching could help is parallel structure which will be discussed in Section 4.5. A grammatically correct way to represent ({Clyde} {Yang} {Wesley}) is ”Clyde, Yang and Wesley” or ”Clyde, Yang, and Wesley”. However, people could also understand ”Clyde Yang and Wesley are friends” or ”Clyde, Yang, Wesley are friends”. Partial matching could help the system match ”Clyde Yang and Wesley” with the parallel structure construction and gets ({Clyde} {Yang} {Wesley}).

We can see partial matching can be very powerful and useful. Unfortunately, we do not have enough time to dig deeper in this field. The rest of the thesis will still mainly discuss perfect match and leave the next step of partial match to future work.

3.4 Constructor

Constructor acts as the main function for a single input text. The Constructor takes in a raw text, a context, and a syntax tag list, and outputs all possible Scone elements, representing the semantics, along with their existing contexts. The input context is used for modifiers as discussed in Section 2.3. The logic is pretty straightforward: Constructor is recursively called in variable match function as showed in Algorithm 1. Therefore, if we have a modifier to that variable, the Constructor can take in the modified context and update it. This part is pretty separate from the main logic in Constructor, details can be found in the original Lisp implementation of the NLU engine. As for now, the remaining discussion of the Constructor will focus on how it manages the text and constructions and coordinates with the Matcher.

Algorithm 3 Constructor

```
$T$ is the input text
$Tgl$ is the syntax tag list, nil if not specified
procedure CONSTRUCTOR($T$ $Tgl$)
    for $CT$ in construction forest do                    ▷ $CT$ is a construction tree
        reset context
        collect TREE CONSTRUCTOR($CT$ $T$ $Tgl$)
    end for
    reset context
end procedure

procedure TREE CONSTRUCTOR($CT$ $T$ $Tgl$)
    $RC \leftarrow$ the root of the $CT$
    $Pl, Cl, Tg, A \leftarrow$ pattern, variable constraints, syntax tag and action of $RC$
    if ($Tgl$ is null) or ($Tg$ in $Tgl$) then
        $MRl \leftarrow$ MATCHER($Tl$ $Pl$ $Cl$)
        for $v$, $ctx$ in $MRl$ do      ▷ the Matcher result is a list of variable value and context
            update context to $ctx$
            $root \leftarrow$ apply $A$ on $v$
            $ctx' \leftarrow$ current context
            $child \leftarrow$Nil
            for $ST$ in children of $CT$ do
                reset context to $ctx'$
                $child$ append TREE CONSTRUCTOR($ST$ $T$ $Tgl$)
            end for
            if $child$ is not null then
                collect $child$
            else
                collect $root$, $ctx$
            end if
        end for
    end if
end procedure
```

The objective of the Constructor is to check and apply all the constructions with the specified syntax tag. As discussed earlier in Section 2.5, constructions are saved in a forest structure to preserve their hierarchy. The Constructor needs to loop through every tree and collect all the results it could get from the trees. The Tree Constructor function takes in the given tree and does

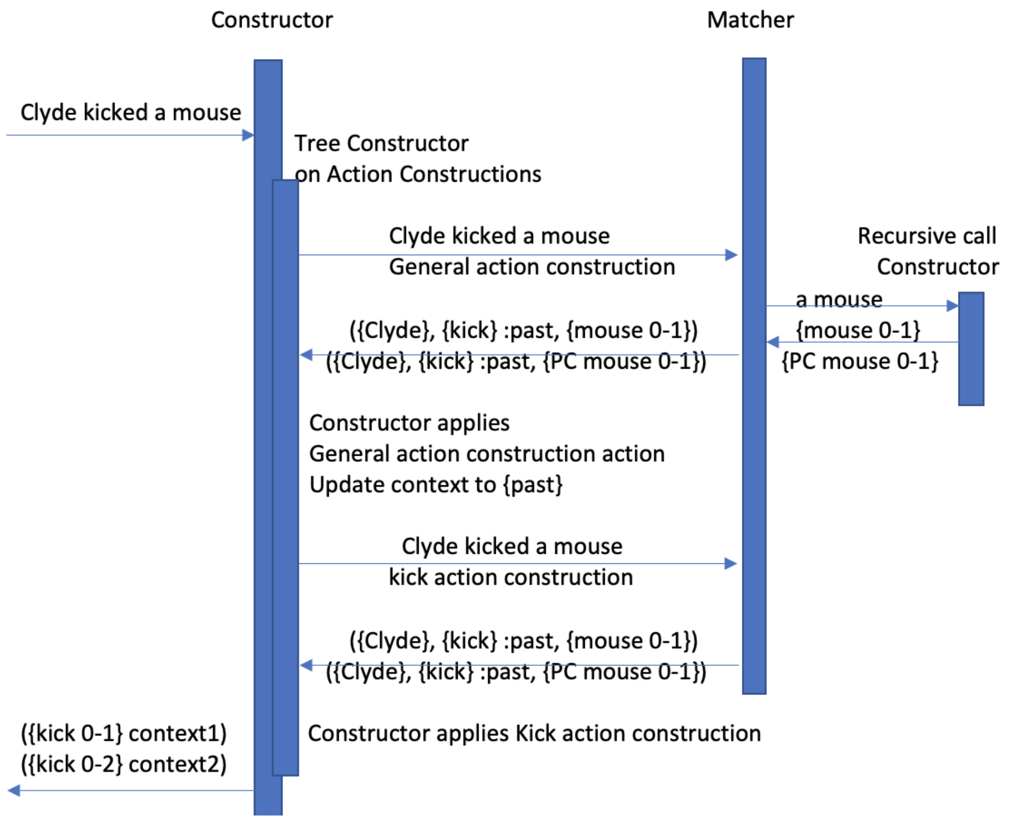


Figure 3.4: Constructor and Matcher example

a depth-first search for matched constructions. Due to the way of building the construction tree, the text could match the children constructions only if it could match with the root construction. Additionally, after it applies the root construction, the context is changed for the children constructions so that parent semantics could affect the semantics of children. Finally, it will collect the result if it cannot go deeper so that the returned semantics are constructed from the least general patterns.

Now, I'll show a full example of how Constructor and Matcher interact as shown in Figure 3.4. The Constructor takes in ”Clyde kicked a mouse” and tries to match with the Transitive Action constructions (the example in the hierarchy section 2.5). The Constructor will first try to match with the General Transitive Action Construction and apply the semantic action to update the context to {past}. Then the constructor will check the children of General Action Construction and find Kick Action Construction match. The Matcher will recursively call the Constructor on ”a mouse” to get two semantics an individual animal mouse {mouse 0-1} and an individual computer mouse {PC mouse 0-1}. After getting the two possible values of the variables, the constructor will apply the semantics on both of them and get two new kick elements {kick 0-1} and {kick 0-2}, where the action object of {kick 0-1} is {mouse 0-1} and action object of {kick 0-2} is {PC mouse 0-1}. Finally, the Constructor cannot go deeper in the construction tree and there are no matches in other trees, so it will return {kick 0-1} and {kick 0-2} with their corresponding

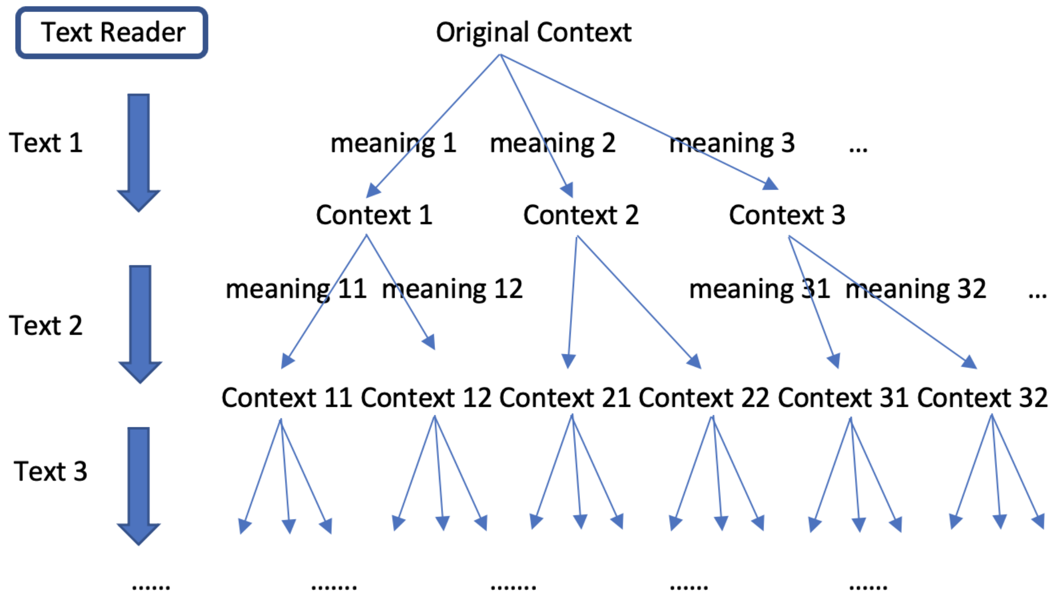


Figure 3.5: Tree structure of context in NLU

context.

3.5 Core Engine

The purpose of the core NLU engine is to make the system able to understand a sequence of input texts based on the previously built Constructor. Note the Constructor actually collects every possible result of applying construction rules, however when we have a large number of texts, it would be very costly in terms of time and space efficiency. Since different understanding of one piece of text might lead to a different context, and a different context would give a different understanding of future texts, the understanding of a sequence of texts forms a tree structure shown in Figure 3.5.

We can see that the tree could grow extremely big if we have lots of input text with ambiguous meanings. Therefore, I decide to adopt a guess and go on mechanism, which means not to collect every possible result but take one result at each time to continue and save all the other results for a future look back if the chosen one does not make sense for future texts. Figure 3.6 demonstrates the idea.

Basically, every time the Text Reader reads a new input text, it will call the Constructor to see if the text has a valid meaning in the current context. If in the current context, the Constructor gives some results, it means the input text makes sense in the current context. Then the system will take the first result, update the context accordingly and save the unused results.

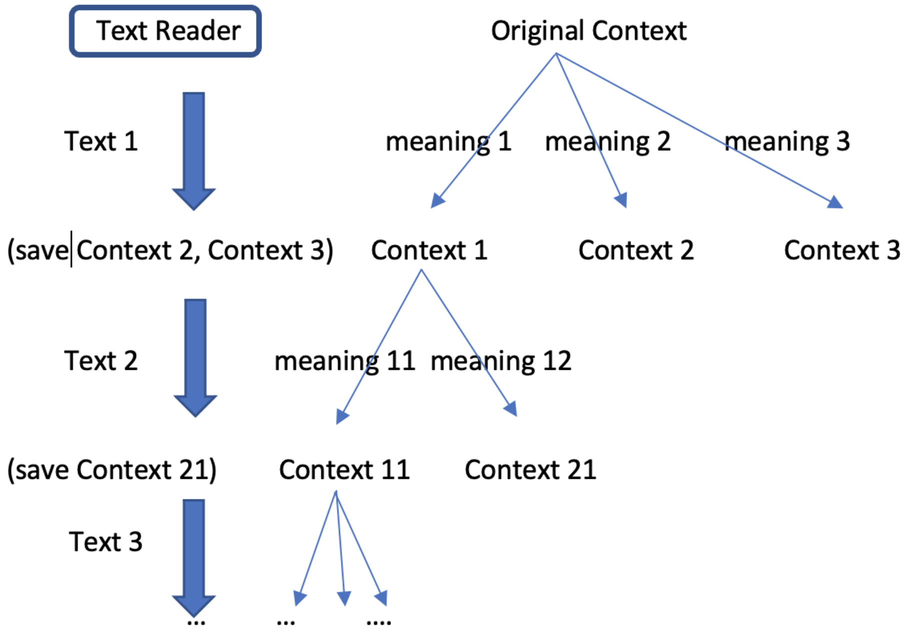


Figure 3.6: Guess and go on

However, if the Constructor gives a null result, that means the text does not make sense in the current context, so we need to do backtracking. The system will go to the previous text and choose a new unused result and context to see if this path could make the future text make sense. The backtracking process will check the unused semantics of the texts in order until it finds a track where the following texts make sense as shown in Algorithm 4.

For example, the Text Reader first takes in ”Yang has a mouse”, the constructor will generate two semantics: either Yang has an animal mouse or a computer mouse. The Text Reader will take the result of animal mouse, store the result of computer mouse and go on. Then the Text Reader takes a new input ”the mouse is a tool”. It will first try to construct under current context while creating an is-a link between animal mouse and tool will cause an error in Scone, so the system will realize current context does not make sense. Therefore, it will backtrack to the previous unused result which is the computer mouse. Now under the computer mouse context, ”the mouse is a tool” makes sense and the Text Reader will save this new path and discard the animal mouse result.

Such guess and go on mechanism intuitively meets how human read long paragraphs. We definitely cannot process and understand the entire passage at once. What we do is we read, most likely sentence by sentence, understand each part and move on. If we run into a sentence that does not make sense, we go back to previously read part and try to understand the sentences again, which is exactly how guess and go on work in the NLU engine.

Algorithm 4 Core NLU

```
$TR$ is used for tracking current using path
$RR$ is a list of unused semantics list
procedure BACKTRACK($Tl$)                                ▷ $Tl$ is referring to future texts
    if all texts in $Tl$ make sense in current context then
        return $Tl$
    end if
    if last entry of $RR$ is null then                   ▷ there's no unused result from the last text
        if $TR$ is null then
            return nil
        end if
        append the last saved text in $TR$ to $Tl$
        remove the last entry of $RR$ and $TR$
        return BACKTRACK($Tl$)                          ▷ the algorithm backtracks one more step
    end if
    extract the first ($semantic$, $ctx$) from the last entry of $RR$
    remove the extracted entry from $RR$
    if $Tl$ make sense under $ctx$ then
        set context to $ctx$
        store ($semantic$, $ctx$) into $TR$
        return $TL$
    else
        return BACKTRACK($Tl$)
    end if
end procedure

procedure TEXT READER($T$)
    $c \leftarrow$ CONSTRUCTOR($T$)
    if $c$ not null then                                 ▷ there are valid meaning for the input text
        store $c[0]$ in $TR$
        store $c[1:]$ in $RR$
        update context according to $c[0]$
        return $c[0]$
    else
        $unR \leftarrow$ BACKTRACK([$T$])
       ▷ backtrack function will backtrack to a state where all future texts will make sense and
returns the unread texts
        if $unR$ is null then
            return nil
        else
            call TEXT READER sequentially on every element of $unR$
        end if
    end if
end procedure
```

Chapter 4

CxG Implementation

4.1 Overview

In this chapter, I will go over the example construction implementation in the system. The purpose is to make the system able to understand simple sentences and give Scone a new way to input knowledge. These implementation mostly focus on dealing with ”meaningful fragments” of sentences, eg. noun phrases, complex verbs, etc, so that the system prototype could get working and at meantime, set a good foundation for understanding complex sentences in the future. I will first elaborate on some details in the implementation to demonstrate how to use the whole system. Then, I will provide examples to show the result of the algorithms described in Chapter 3. Additionally, these examples can demonstrate how Scone is used to save the semantics of the natural language.

4.2 Noun Phrase

Noun phrases are the fundamental constructions in a grammatically correct sentence. Having well-defined noun phrase constructions is essential for further implementations. I consider the following four types: singular noun phrases (like ”an elephant”), plural noun phrases (like ”some elephants”), referral noun phrases (like ”the elephant”), and possessive noun phrases (like ”Clyde’s elephant”). While singular and plural noun phrases aim to create new Scone elements to represent what the phrase is referring to, referral noun phrases are considering the phrases that refer to previously mentioned elements. Finally, possessive noun phrases are referring to noun with possessive determiners, which is a relatively complex case.

4.2.1 Singular Noun Phrases

The most common singular noun representation is in the form ((“a” “an”) ?x) where ?x is a singular form noun. This noun phrase refers to an individual noun element that is of ?x type. Therefore, the semantics of the phrase is intuitively (new-indv NIL ?x). Additionally, we need to add the newly created element to the referral context so that the element could be referred to later.

New Singular Noun Phrase	
Pattern	(”a” ”an”) ?x
Variable Constraints	(?x :noun :type)
Return Tag	:noun
Semantics	?y ← (new-indv NIL ?x) referral context add ?y return ?y

Figure 4.1: Singular noun phrase construction

For example, when the user calls (**read-text** ”an elephant”), the Constructor will apply the Matcher on ”an elephant” and the singular noun phrase construction. ?x variable will be matched with the type element {elephant}, which will be plugged into the construction action. A new individual elephant {elephant 0-2702} will be created and added to referral context.
If the user calls (**read-text** ”a mouse”), note the system will match ?x with both {mouse} (animal mouse) and {computer mouse}. As the way Text Reader works, it will output a new individual animal mouse {mouse 0-2711} with the context changed to {general 0-2710}, a children of the previous {general} context. While in the result record, the system stores a new computer mouse individual {computer mouse 0-2714} with its existing context {general 0-2713}.

4.2.2 Plural Noun Phrase

While singular forms can be easily represented as an individual node in Scone, plural forms require a bit of extra work. A set of objects are represented as type nodes in Scone but in natural language people usually also quantify the number of the objects. The number could be precise like ”two elephants”, ”a dozen of eggs” or it also could be inaccurate like ”some elephants” or ”many elephants”. Therefore, we need a way to represent these types of determiners.

First, I assume countable nouns are tangible objects and define a new type role {count} for {tangible} :(**new-type-role** {count} {tangible} {number}). In Scone, defining a type-role node is basically creating a has link where in this case we specifies a {tangible} element has a {count} which needs to be a {number} type node. Therefore, we could use {count} to specify the size of the set of objects.

Then, I need to define new {number} nodes.
Distinguishing precise and imprecise numbers is also not a trivial task in natural language understanding. For example, when we say ”There are a hundred seats in the club”, a hundred may be imprecise, while if we say ”There are one hundred and two seats in the club”, one hundred and two is most likely to be an exact number. However, in the scope of this project, I am dealing with the simple case that numbers are assumed to be accurate.

New Plural Noun Phrase	
Pattern	?x ?y
Variable Constraints	(?x {number}) (?y {tangible} :noun :type)
Return Tag	:noun
Semantics	?z ← (new-type NIL ?y) (x-is-the-y-of-z ?x {count} ?y) referral context add ?z return ?z

Figure 4.2: Plural noun phrase construction

For precise numbers, I create a new individual node and link this node with the number node by an eq-link. For example, to represent ”a dozen”, I first create a new node (**new-indv** “a dozen” {exact number}), and then using equal link to link the node with {12}: (**new-eq** {a dozen of} {12}).

To represent imprecise expressions like “many” or “some”, I assume these expressions are equivalent with an integer range. For example, ”some” means at least two, so it’s a range whose lower bound is two. Consequently, I created a new type {integer range} and it has a {lower bound} and a {upper bound}:

(**new-type** {integer range} {inexact number})
(**new-type-role** {lower bound} {integer range} {integer})
(**new-type-role** {upper bound} {integer range} {integer})

For example, to express “some” in Scone, we first create a new {integer range} type node (**new-indv** “some” {integer range}). Then, we can specify the lower bound: (**x-is-the-y-of-z** 2 {lower bound} {some})

Finally, after we settled the representation of plural nouns, we could define the construction as Figure 4.2. The pattern is a {number} type node ?x followed by a {tangible} type node ?y. As discussed above, the semantics of the construction is to first create a new type node as ?y’s children. Then the system will specify the {count} of the new type node to be ?x and add the new node to the referral context.

For example, when the user calls (**read-text** ”some elephants”), the Constructor will apply the Matcher on ”some elephant” and the plural noun phrase construction. ?x variable will be matched with the element {some} that we just created and ?y variable will be matched with {elephant}. The construction action will return a new type node {elephant 0-2706} to represent the semantics of ”some elephants”. Now if we look into Scone and ask (**the-x-of-y** {count} {elephant 0-2706}), it will return {some}; if we ask the lower bound of the count (**the-x-of-y** {lower bound} (**the-x-of-y** {count} {elephant 0-2706})), Scone will output {2}, which is exactly what

Referral Noun Phrase	
Pattern	(”the”) ?x
Variable Constraints	(?x :noun :type)
Return Tag	:noun
Semantics	loop for ?y in referral context when (simple-is-x-a-y? ?y ?x) add ?y to referral context return ?y

Figure 4.3: Referral noun phrase construction

we desired.

Similarly, if the user calls (**read-text** ”two elephants”), the system will output a new elephant type node {elephant 0-2697}, where in Scone (**the-x-of-y** {count} {elephant 0-2697}) is {two} and (**is-x-eq-y?** {two} {2}) returns :YES.

4.2.3 Referral Noun Phrase

While the singular and plural noun phrases constructions create new nodes, referral noun phrases are referring to previously created nodes that are saved in referral context. The most common referral noun representation is in the form ((“the”) ?x) where ?x is a type node. This noun phrase refers to a previously referred individual noun element that is of ?x type. Therefore, the semantics of the phrase is intuitively to loop through the referral context and return the element that is of ?x type. Finally, we need to re-add this element to the referral context so that this element is back on the top of the stack.

For example, say the user initially calls (**read-text** ”Clyde is an elephant”) and a new element {Clyde} is already added into the referral context. Then, when the user calls (**read-text** ”the elephant”), the Constructor will apply the Matcher on ”the elephant” and the referral noun phrase construction. ?x variable will be matched with the type element {elephant}, and then the construction action loops through the referral context, gets {Clyde} and since (**simple-is-x-a-y?** {Clyde} {elephant}) returns true in Scone, the Constructor will return {Clyde} as the semantics of ”the elephant”.

4.2.4 Possessive Noun Phrase

Possessive noun phrase, as the name suggests, refers to the pattern of a possessive determiner followed by a noun. In general, there could be two different possessive noun phrases: possessive determiner and a type node like ”his cat” or possessive determiner and a type-role node like ”his

Possessive Type-role Noun Phrase	
Pattern	?x ?y
Variable Constraints	(?x :noun)
	(?y :noun :type-role)
Return Tag	:noun
Semantics	if (:possessive not in (modifier ?x)) return NIL
	(new-is-a ?x (context-ele ?y))
	loop for ?z in referral context
	when (simple-is-x-a-y? ?z (parent ?y))
	and (?z in (list-all-x-of-y ?y ?x))
	add ?z to referral context
	return ?z
	——-if nothing returned——-
	?z ← (new-indv nil (parent ?y))
	(x-is-a-y-of-z ?z ?y ?x))
	add ?z to referral context
	return ?z

Figure 4.4: Possessive type-role noun phrase construction

friend". For the purpose of demonstration, I will mainly focus on the possessive type-role node here since it is more different from the other types of noun phrases, while possessive type node construction could be defined similarly.

The pattern of possessive type-role is pretty obvious" ?x ?y, where ?x is a noun and ?y is a type-role node. The first thing to do in construction action is to check the modifier tags of ?x needs to include :possessive. However, what makes possessive type-role complicated is that in some cases it refers to a previously referred element, for example, "Wesley and Yang are friends. Wesley gave his friend a gift.", "Wesley's friend" is most likely referring to Yang; while in some other cases, for example, "Mark Zuckerberg and his roommate are the founders of Facebook", here "his roommate" is a new individual that is the roommate of Mark Zuckerberg. To make things easier, I make the semantics follow the logic that if there is a previously referred element in referral context that is the ?y of ?x, then we return it, otherwise, it will create a new individual. Note in the semantics part in Figure 4.4, it also checks the parent-element and context-element of the type-role node ?y. To demonstrate their usage in Scone, say we have a type-role node {pet}, which is defined by (**new-type-role** {pet} {person} {animal}). The definition implies that a {person} element can have {pet} and {pet} needs to be an {animal} element. In Scone, the parent-element of {pet} would be {animal} and the context-element would be {person}.

For example, the user calls (**read-text** "Yang's pet") and tries to match the text with the possessive type-role construction. ?x will be matched with {Yang} with a modifier tag :possessive and ?y will be matched with the type-role node {pet}. The semantics action will first create a new is-a link between {Yang} and {person}, which is the context element of {pet}. This action guarantees that only a person could have a pet, which is a hidden information in the text. If there are no referral context, the system will create a new individual {animal 0-2946}, since {animal} is the parent element of {pet}. Finally, it claims {animal 0-2946} is a {pet} of {Yang} in Scone, adds {animal 0-2946} to referral context and return it. If in the previous context, we know ”Yang’s pet is a dog” and {dog 0-2951} added to referral context, then the construction action will check {dog 0-2951}, previous referred, is an {animal} and is a {pet} of Yang. Therefore, it returns {dog 0-2951} without creating a new animal element.

Note there are some special cases where people use type node to represent the meaning of a type-role node. For example, when people say ”my man” it actually means ”my friend” or ”my pal”, and it does not mean that I am in possession of a man. We could take use of construction hierarchy to specify these slang with their specific meanings.

4.3 Adjectives

In natural language, using adjectives is the most common way to further describe and decorate noun phrases. Scone has its own way to store information about adjectives and we need to deal with it carefully since there are various ways of using adjectives in noun phrase constructions and static description constructions. I am going to use color as an example to demonstrate three common constructions that use adjectives.

First, we need to figure out how to represent adjectives in Scone. In Scone, adjectives representation normally combines with the object. For example, “red” is represented as {red thing}. The tricky thing here is ”the color of A is red” should have the same meaning as ”A is red”. When we say something is red, it means the predominant color of this thing is red. Therefore, we need to set up a connection between the noun color {red} and the adjective {red thing}.

The following lines helps to strictly define color in Scone:
(**new-type** {color} {tangible})
(**new-type** {colored thing} {tangible} :english ’(:no-iname :adj ”colored”))
(**new-type-role** {predominant color} {colored thing} {color} :english ’(”color”))

Then we could set up the connection between color red and adjective red:
(**new-type** {red thing} {colored thing} :english ’(:no-iname :adj ”red”))
(**new-indv** {red} {color})
(**x-is-the-y-of-z** {red} {predominant color} {red thing})
Now, when we say something is a {red thing} in Scone, it also tells the {predominant color} of it is {red}. Once these are set up, we can define adjectives related constructions.

The first type of construction is using adjectives directly describing the noun, for example, sin-

New Singular Noun Phrase with Adjective	
Pattern	(”a” ”an”) ?x ?y
Variable Constraints	(?x :adj) (?y :noun :type)
Return Tag	:noun
Semantics	?z ← (new-indv NIL ?y) (new-is-a ?z ?x) referral context add ?z return ?z

Figure 4.5: Singular noun phrase construction with adjective

Being verb with Adjective	
Pattern	?x (”is” ”are”) ?y
Variable Constraints	(?x :noun) (?y :adj)
Return Tag	:relation
Semantics	return (new-is-a ?x ?y)

Figure 4.6: Static adjective description

gular noun phrase with adjective construction shown in Figure 4.5. The pattern is ((“a” “an”) ?x ?y) where ?x is an adjective and ?y is a type noun. Basically what the construction action would do is first apply the action of singular noun structure, creating a new individual node of ?y type, and then say the new individual is a ?x. For example, the system takes in “a red apple”, the matcher will match ?x with {red thing} and match ?y with {apple}. Then the construction action will call (**new-indv** NIL {apple}) to generate {apple 0-2333}. To represent the semantics of adjective red, the system will add a new is-a link in Scone: (**new-is-a** {apple 0-2333} {red thing}).

The second type of construction is described using linking verb and the most common way is to use being verb as shown in Figure 4.6. The semantics is more intuitive: creating an is-a link in Scone. For example, the system takes in ”apples are red”, and the Matcher matches ?x with {apple} and ?y with {red thing}. Then the action will create the is-a link between {apple} and {red thing} in Scone.

The third type of construction is using type-role directly, for example, “the predominant color of apple is red”. The intuitive semantics would be to specify the {predominant color} of {apple} to be {red} and as we discussed above, such action is equivalent with stating {apple} is a {red thing} in Scone. The detailed construction of type-role related static description will be elaborated in the next section.

Being verb with individual node	
Pattern	?x (”is” ”are”) ?y
Variable Constraints	(?x :noun) (?y :noun :indv)
Return Tag	:relation
Semantics	return (new-eq ?x ?y)

Create new is-a	
Pattern	?x (”is a”) ?y
Variable Constraints	(?x :noun) (?y :noun :type)
Return Tag	:relation
Parent	Being verb with individual node
Semantics	return (new-is-a ?x ?y)

Figure 4.7: State of being verb based description

4.4 Static Description

Static description is very suitable for graph-based knowledge representation since it describes a static relationship between elements. One of the primary goals of this project is to build well-covered constructions for static description so that Scone could have a way to input knowledge. I consider the following four cases in static description: state of being verb-based description, static type-role description, has relation description and stative verb-based description.

4.4.1 State of Being Verb Based Description

Using being verbs ”is” and ”are” is the common case static description. The basic pattern is (?x (”is” ”are”) ?y), where ?x is a noun element and ?y could be an individual noun node, a type noun node or an adjective (which is already discussed in the previous section). Different types of ?y node have different constructions and semantics.

When ?y is a type node, for example ”elephants are animals”, then the Matcher will match ?x with {elephant} and ?y with {animal}. Consequently, the construction action would be to add a is-a link from ?x to ?y, where in this case (**new-is-a** {elephant} {animal}).

When ?y is an individual node, for example say there is a previously mentioned elephant {elephant 0-3141} and the engine takes in an input “Clyde is the elephant”, then the Matcher will match ?x with {Clyde} and ?y with {elephant 0-3141}. We can see the semantics of this sentence is basically telling {Clyde} and {elephant 0-3141} are referring to the same thing. Therefore, the construction action should be creating an equal link between ?x and ?y in Scone.

Implicit type-role description	
Pattern	?x ("is the") ?y
Variable Constraints	(?x :noun) (?y :type-role)
Return Tag	:relation

Semantics	?z ← (context-element ?y)
	loop for ?w in referral context
	  when (simple-is-x-a-y? ?w ?z)
	  return (x-is-the-y-of-z ?x ?y ?w)

Figure 4.8: Static type-role friend construction

In addition, there is a specific pattern of being verb with individual node that the engine could do differently: (?x ("is a" "is an")?y), where ?x is a noun and ?y is a type noun node. In this case, the pattern could fall into the case (?x ("is") ?y) and ?y is an individual node. Then the engine will apply singular noun phrase construction, generate a new individual node and connect ?x with an eq-link. However, the output of such construction action is not intuitive enough. A more direct action is just call (**new-is-a** ?x ?y). Therefore, as shown in Figure 4.7 we could make this construction a child of the above case so that if the text match with the Create New Is-a construction it will replace the output of Being Verb with Individual Node construction. For example, when we say, "Clyde is an elephant", the Matcher will first match with Being Verb with Individual Node construction and apply the corresponding action, then it will match with the children Create New Is-a construction, apply the action (**new-is-a** Clyde elephant) and finally output the is-a link as the result instead of an eq link.

4.4.2 Static Type-role Description

As we mentioned before, in Scone, type role node is used to describe the "A of B" relation. When defining new type-role element A, the user needs to specify the type of B and the type of "A of B", which directly fit into the pattern (?x ("is a") ?y ("of") ?z), where ?y is a type-role node. Similar patterns that fall into this category are (?x ("is the") ?y ("of") ?z), (("the") ?x ("is" "are") ?z), etc. The action of these constructions could be formed by directly calling the **x-is-a-y-of-z** and **x-is-the-y-of-z** methods. For example:
"Yang is a friend of Wesley" ⇒ (**x-is-a-y-of-z** {Yang} {friend} {Wesley})
"The roommate of Wesley is Yang" ⇒ (**x-is-the-y-of-z** {Yang} {roommate} {Wesley})
"The predominant color of apple is red" ⇒ (**x-is-the-y-of-z** {red} {predominant color} {apple})

However, in natural language, sometimes people might use type-role but do not explicitly write what "of B" is if it is previously mentioned. For example, if the company {Facebook} is previously mentioned, people can directly say "Mark Zuckerberg is the founder", assuming others could understand it means "the founder of Facebook". The construction of Implicit type-role construction is shown in Figure 4.8. The matcher will match ?x with {Mark Zuckerberg} and ?y with

Implicit type-role description	
Pattern	?x (”are”) ?y
Variable Constraints	(?x :noun :list) (?y :type-role)
Return Tag	:relation
Semantics	if len(?x) $\leq$ 1: return NIL loop for i from 0 to (len(?x) - 1) loop for j from i+1 to (len(?x) - 2) collect ’((x-is-a-y-of-z ?x[i] ?y ?x[j]) (x-is-a-y-of-z ?x[j] ?y ?x[i]))

Figure 4.9: Static type-role friend construction

{founder}. The construction action will first check the context-element of {founder} in Scone and gets {organization}. Then the action looks into referral context and finds {Facebook} ,which is an {organization}, and returns (**x-is-the-y-of-z** {Mark Zuckerberg} {founder} {Facebook}).

Finally, there’s another way in natural language where type-role are commonly used: (?x (“are”) ?y), where ?x is a list of nouns, which is constrained by :list keyword, and ?y is a symmetric type-role (meaning in “A of B” the type of B is the same as the type of “A of B”). For example, {friend}, {roommate} are all symmetric since both its context-element and parent-element are {person}. Then this construction means every pair of nouns in ?x could fit into “A is a ?y of B”. For example, when we say “Wesley and Yang are friends”, the Matcher will match ?x with ’({Wesley}, {Yang}) and ?y with {friend}. The construction action would be (**x-is-a-y-of-z** {Wesley} {friend} {Yang}) and (**x-is-a-y-of-z** {Yang} {friend} {Wesley}).

4.4.3 Has Relation

The composition relationship is also a major part in static description. The most common pattern is (?x (”has” ”have”) ?y ?z), where ?x ?z are two nouns and ?y is a {number}. The way Scone represent this has relation is still type role. As we describe before, type role node tends to represent ”A of B”, while this actually implies B has A. Therefore, when the user defines a new type-role, say (**new-type-role** {pet} {person} {animal}), Scone creates the new node {pet} as a children of {person} and connect a has-link between {person} and {pet}.

For example, when the user calls (**read-text** ”Clyde has four legs”) and the Matcher tries to match with the has relation construction, ?x will be matched with {Clyde}, ?y matched with {four} (which is equal to the number node {4}) and ?z with {leg}. The construction action will create a new type-role {leg 0-2813} representing the legs of Clyde and specifying the quantity of it is {4}.

Has relation description	
Pattern	?x (”has” ”have) ?y ?z
Variable Constraints	(?x :noun) (?y number) (?z :noun :type)
Return Tag	:relation
Semantics	?w ←(new-type-role nil ?x ?z :n ?y) add ?w to referral context return ?w

Figure 4.10: Has relation description

There are two benefits of using type-role to represent this relation: 1. The creation of this new type-role, in Scone a has-link between Clyde and leg is internally constructed. 2. Due to this type-role is added to the referral context, the system will be able to understand “Clyde’s legs” and interpret it as {leg 0-2813} in future text. However, since the has-link functionality is changing and under review right now in Scone, there’s no simple way to resolve the case where such type-role or a has link has already been constructed and the engine needs to alter or change the link. This would require some future work when the functionality of the has-link is fully settled.

4.4.4 Stative Verb

Stative verbs are commonly used to describe the state of being. Stative verbs could be divided based on their semantics: verbs of perception, verbs of cognition, emotion, etc [9]. Typically, stative verbs, like ”love”, ”hate”, ”believe”, are used to describe opinions, beliefs, and emotions. These verbs could be represented as relations in Scone.

Scone allows the user to create new relations and then to instantiate these using statement links. For example, we could define ”hate” as a one-way relationship between {animal} and {thing}. When the system takes an input ”Jerry hates elephants”, the construction action would be to create a new statement link that instantiates the {hate} relation, where in this case, (**new-statement** {Jerry} {hate} {elephant}). Now say that the system takes in a new text ”Clyde is an elephant”, and we go to Scone and ask if Jerry hates Clyde: (**statement-true?** {Jerry} {hate} {Clyde}), Scone will return True which meets our expectation.

4.5 Parallel Structure

Parallel structure is very commonly used in natural language. To represent the structure, we can use a list to combine each component. For example, “apple, banana, and grape” should be represented as (list {apple} {banana} {grape}).

Noun phrase parallel with and	
Pattern	?x (", and", "and") ?y
Variable Constraints	(?x :noun) (?y :noun)
Return Tag	:noun
Semantics	return (list ?x ?y)

Noun phrase parallel with comma	
Pattern	?x (",") ?y
Variable Constraints	(?x :noun) (?y :noun :list)
Return Tag	:noun
Semantics	if (len(?y) $\geq$ 2) return (append (?x) ?y)

Figure 4.11: Noun phrase parallel structure

In most cases, when a variable is a list of objects, we should apply the construction rule on the objects separately and collect all the results. For example, when we have “Clyde and George are elephants”, it means “Clyde is an elephant” and “George is an elephant”. So applying construction grammar on it should return two is-a links.

However, there are also cases the variable itself should be a list. For example the Static type-role friend construction is shown at Figure 4.8. When we say “A, B, C, ... and D are friends”, we want to have the (list A B C ... D) as a single element since we want to add new relation to every pair of elements in the list.

To distinguish the above two cases, the keyword :list, as we discussed in 2.2.2, is used as a variable constraint to specify that the variable should be treated as a list in the construction action. With :list, we can strictly define the parallel structure A, B, ... and D.

The tricky thing about the parallel structure is that the pattern could go infinitely long with commas but always ends with an ”and”. Phrases like ”Yang, Wesley, Tony” is not grammatically correct (but it is understandable with partial matching). Therefore, first, we need to recursively define this pattern and second, we use :list in the parallel with comma construction so that we can constrain the length of ?y.

For example, if the system takes in ”Yang, Wesley, and Tony”, the tokenizer will break the sentence into ’(”Yang” ”,” ”Wesley” ”,” ”and” ”Tony”) and the Matcher will try to match it with the Noun phrase parallel with comma construction, where ?x is matched with {Yang} and ?y is matched through recursively calling the constructor on ”Wesley, and Tony”. Then ”Wesley, and Tony” will be matched with Noun phrase parallel with and construction, and get an output

location prepositional phrase for relation	
Pattern	?x (”at” ”in” ”on”) ?y
Variable Constraints	(?x :relation) (?y place :noun)
Return Tag	:relation
Modifier	?x (new-context nil ’(current-context {place}))
Semantics	(new-eq (context-element ?x) ?y) return ?x

Figure 4.12: Location prepositional phrase for relation

of ’({Wesley} {Tony}). Then the action of Noun phrase parallel with comma construction will finally output ’({Yang} {Wesley} {Tony}).

On the other hand, if the system takes in only ”Yang, Wesley”, the Matcher will match it with Noun phrase parallel with comma construction, where ?x is {Yang} and ?y is ’({Wesley}) due to the :list keyword. Then if we plug in ?x and ?y into the construction action, the length of ?y is 1 which is smaller than 2. Therefore, it will return null and further tell the system this is not a valid construction match.

4.6 Prepositional Phrase

A prepositional phrase is often used to modify other nouns, actions or relations and it usually consists of a preposition and other words to modify the object. For example, in the sentence ”Clyde kicked a ball on Monday”, ”on Monday” is a time prepositional phrase that is used to modify the time of the action. Note the purpose of the prepositional phrase is exactly the same as the usage of modifiers in constructions. To demonstrate how modifier is used, I define the location prepositional phrase for relation construction as shown in Figure 4.12.

As we can see, the pattern needs to be (?x :relation) (”at” ”in” ”on”) (?y place :noun), where ?y is the location used to modify ?x. For example, the system takes in ”elephants are grey in Africa” and the Matcher tries to match with the Location prepositional phrase for relation construction. To get the value of ?x, the system will first modify the context to a new dummy context that inherits current context and {place}, say {general 0-2755}. Then by recursively calling Constructor on ”elephants are grey”, the system will get a relation node {Is-A 0-2758} under the context {general 0-2755} as the semantics of ”elephants are grey” and the value for ?x. ?y is matched with {Africa} and then the construction action will create an eq-link between {general 0-2755} and {Africa}. And finally return {Is-A 0-2758} as the semantics of the whole sentence. Now if we look into Scone, (**context-element** {Is-A 0-2758}) will return {general 0-2755} and (**is-x-eq-y?** {general 0-2755} {Africa}) will give :YES.

Similarly, we can define the prepositional phrase constructions that modifies time, location, etc. for nouns, relations, actions.

Chapter 5

Conclusion

5.1 Major Contributions

In this thesis, we proposed an innovative rule-based natural language understanding system that is built on construction grammar and Scone knowledge base. We first give a well-defined structure for constructions in Lisp and implement a macro function for the users to easily define new constructions by themselves. Additionally, we define some most commonly used constructions so that the system could proceed with simple texts. Then, Matcher and Constructor algorithms are implemented , which support the system to understand different semantics of a single piece of text. Finally, the full system use guess and go on algorithm to understand multiple texts and input the underlying semantics into Scone.

In conclusion, the project gives the following major contributions:

- An innovative way to build an actual natural language understanding system with construction grammar and Scone knowledge base. Construction grammar gives more flexibility to the input text, for example the text does not need to be grammatically correct as long as it matches with a construction rule. Partial matching will further extend the flexibility of the text. Additionally, Scone is also stronger than most of the knowledge graph that were used to store semantics. The context mechanism in Scone also makes it possible to explore different potential meanings of the same text.

- Adopt Guess and Go on mechanism to understand a sequence of texts. Intuitively, when human being reads long paragraphs where some pieces of text could have different semantics, naturally, guess and go on is what we do. If the following texts suggests current context does not make sense, we would go back to the previous texts and guess another meaning.

- The system provides a platform for construction grammar researchers where new constructions could be easily defined and add into the system and then the user could test and see how the new construction work with different texts.

- The system gives a new way to input new knowledge to the Scone knowledge base. I already implemented some of the most commonly used constructions for simple sentences. Then, instead of writing Lisp code to create new knowledge pieces, user could input simple texts to the NLU system and the system will generate the corresponding knowledge in Scone.

5.2 Future Work

Since this is only a one-year long project, there are lots of places that are unfinished and could be further improved. We'd like to provide some future research directions in this project.

Constructions are not equally defined in different construction grammar frameworks. Therefore, to better support linguistic researches, we would want to provide as much flexibility as possible in defining new constructions in the system. In the current construction structure, we would want to support more logic operations in defining variable constraints and consequently, the user would be able to define the construction variable with more specific constraints. Additionally, the system only support context element as modifiers, future work could extend modifier to more forms. For example, :Maybe tag could be another common modifier to modify the level of ambiguity of a piece of text. Finally, as we previously discussed, current construction hierarchy only support single parent structure. However, a better construction hierarchy structure should be able to support multiple parents and multiple children.

Once constructions are better defined in system, we could try to merge with some maturely built construction grammar frameworks. The ultimate goal of an NLU engine is to built a system that is able to learn knowledge from any text input, which requires the constructions for an entire language. However, manually creating the constructions is definitely a tedious task. Therefore, automated creating of construction rules would be an exciting field to explore. For example, there are previous works [2] that provide a metric to learn constructions in Embodied Construction Grammar.

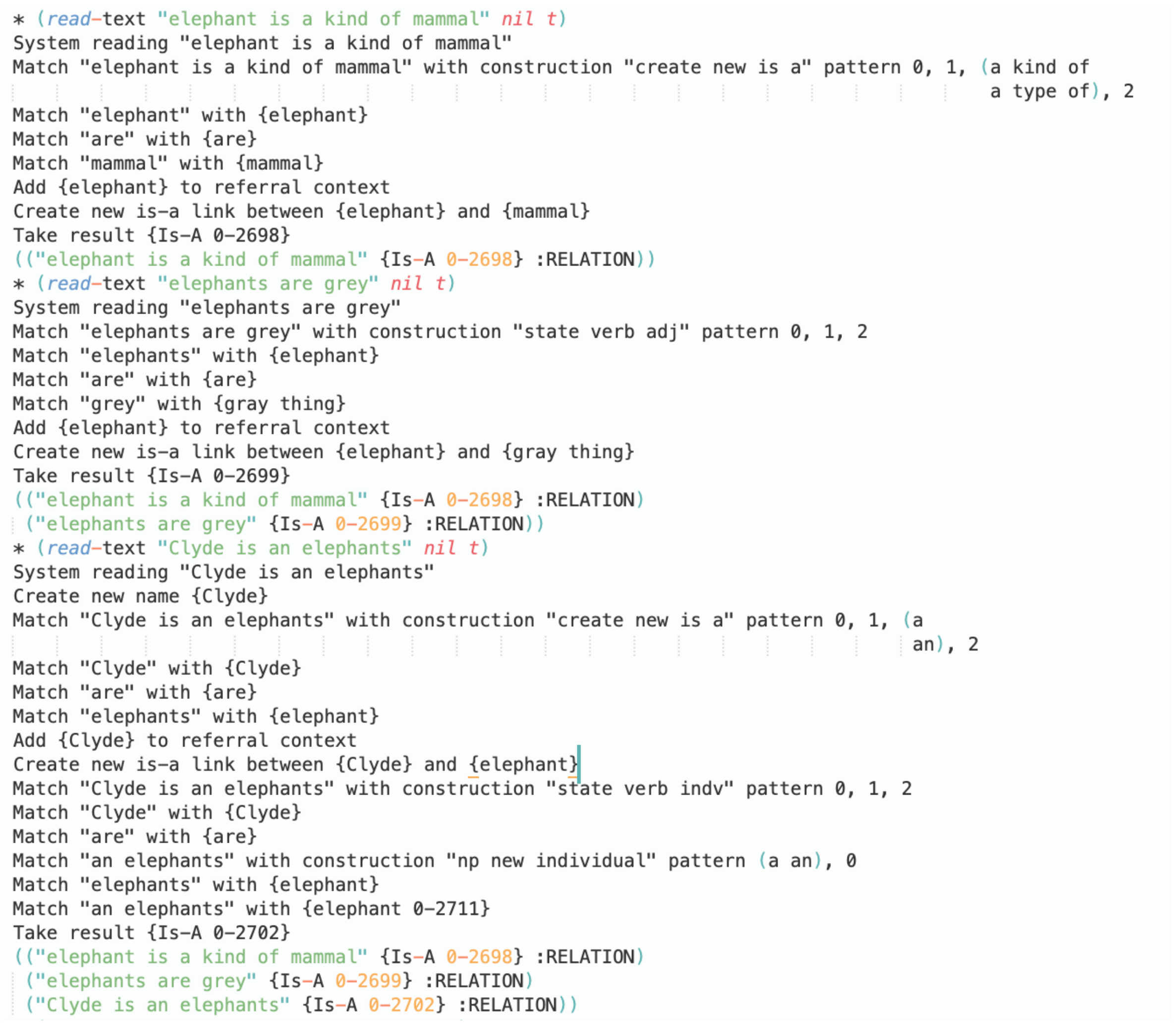
As we discussed in the Matcher section 3.3, if we want to extend the usage to fields with lots of unstructured texts like dialogue, partial matching would be essential. The current partial matching algorithm is relatively naive and further research could be done in this area. For example one could explore scoring metric for partial matching where critical components will have a higher score and missing components will cause a penalty. Then, every text and construction pair will have a matching score and if the score is above a certain threshold, we could apply partial matching between the text and construction.

To further improve the accuracy of the core NLU engine, a possible future work direction is to introduce probability metrics in matching. Intuitively, although words or sentences can have different semantics, some of them are more commonly used or more probable in current context. For example, when we say ”there is a mouse beside my laptop”, it is most probable that the ”mouse” is referring to {computer mouse}. Therefore, we should give {computer mouse} a higher probability than animal {mouse} in this context. Building such probability metric is a pretty complex task since the probability needs to change with different context. However, a well-build probability metric could significantly improve the performance of Guess and Go on algorithm since we could always guess and choose the semantics with the highest probability.

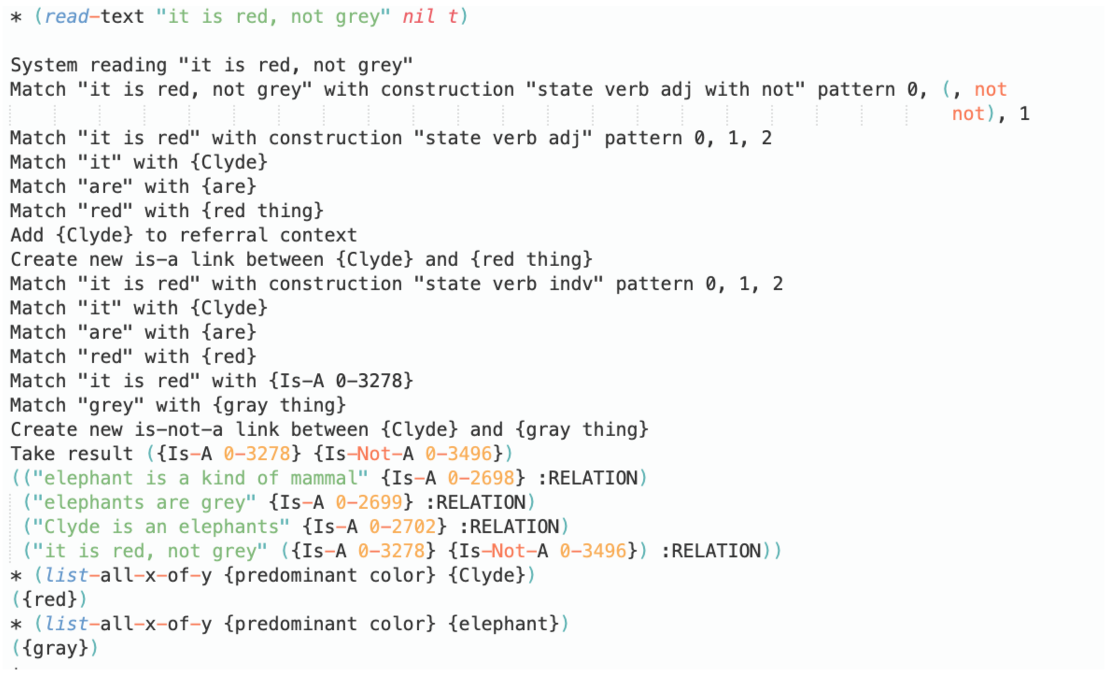
Above are just a few directions that could be done in future research. I fully believe in the potential of Scone and this project aims to give an innovative way of thinking and approaching a real natural language understanding system.

Appendix A

Sample output of Current Implementation



```
* (read-text "elephant is a kind of mammal" nil t)
System reading "elephant is a kind of mammal"
Match "elephant is a kind of mammal" with construction "create new is a" pattern 0, 1, (a kind of
                                                                                        a type of), 2
Match "elephant" with {elephant}
Match "are" with {are}
Match "mammal" with {mammal}
Add {elephant} to referral context
Create new is-a link between {elephant} and {mammal}
Take result {Is-A 0-2698}
(("elephant is a kind of mammal" {Is-A 0-2698} :RELATION))
* (read-text "elephants are grey" nil t)
System reading "elephants are grey"
Match "elephants are grey" with construction "state verb adj" pattern 0, 1, 2
Match "elephants" with {elephant}
Match "are" with {are}
Match "grey" with {gray thing}
Add {elephant} to referral context
Create new is-a link between {elephant} and {gray thing}
Take result {Is-A 0-2699}
(("elephant is a kind of mammal" {Is-A 0-2698} :RELATION)
 ("elephants are grey" {Is-A 0-2699} :RELATION))
* (read-text "Clyde is an elephants" nil t)
System reading "Clyde is an elephants"
Create new name {Clyde}
Match "Clyde is an elephants" with construction "create new is a" pattern 0, 1, (a
                                                                                 an), 2
Match "Clyde" with {Clyde}
Match "are" with {are}
Match "elephants" with {elephant}
Add {Clyde} to referral context
Create new is-a link between {Clyde} and {elephant}
Match "Clyde is an elephants" with construction "state verb indv" pattern 0, 1, 2
Match "Clyde" with {Clyde}
Match "are" with {are}
Match "an elephants" with construction "np new individual" pattern (a an), 0
Match "elephants" with {elephant}
Match "an elephants" with {elephant 0-2711}
Take result {Is-A 0-2702}
(("elephant is a kind of mammal" {Is-A 0-2698} :RELATION)
 ("elephants are grey" {Is-A 0-2699} :RELATION)
 ("Clyde is an elephants" {Is-A 0-2702} :RELATION))
```



```
* (read-text "it is red, not grey" nil t)

System reading "it is red, not grey"
Match "it is red, not grey" with construction "state verb adj with not" pattern 0, (, not
                                                                                     not), 1
Match "it is red" with construction "state verb adj" pattern 0, 1, 2
Match "it" with {Clyde}
Match "are" with {are}
Match "red" with {red thing}
Add {Clyde} to referral context
Create new is-a link between {Clyde} and {red thing}
Match "it is red" with construction "state verb indv" pattern 0, 1, 2
Match "it" with {Clyde}
Match "are" with {are}
Match "red" with {red}
Match "it is red" with {Is-A 0-3278}
Match "grey" with {gray thing}
Create new is-not-a link between {Clyde} and {gray thing}
Take result ({Is-A 0-3278} {Is-Not-A 0-3496})
(("elephant is a kind of mammal" {Is-A 0-2698} :RELATION)
 ("elephants are grey" {Is-A 0-2699} :RELATION)
 ("Clyde is an elephants" {Is-A 0-2702} :RELATION)
 ("it is red, not grey" ({Is-A 0-3278} {Is-Not-A 0-3496}) :RELATION))
* (list-all-x-of-y {predominant color} {Clyde})
({red})
* (list-all-x-of-y {predominant color} {elephant})
({gray})
```

Bibliography

[1] B.K. Bergen and N. Chang. Embodied construction grammar in simulation-based language understanding. *Construction Grammar(s): Cognitive and Cross-Language Dimensions*, 2005. 1.3

[2] Nancy Chang and Olya Gurevich. Context-driven construction learning. *the 26th Annual Meeting of the Cognitive Science Society*, 2004. 5.2

[3] Scott E. Fahlman. Marker-passing inference in the scone knowledge-base system. *International Conference on Knowledge Science, Engineering and Management*, pages 114–126, 2006. 1.2, 1.2

[4] Scott E. Fahlman. Using scone's multiple-context mechanism to emulate human-like reasoning. *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, 2011. 1.2, 1.2

[5] Scott E. Fahlman. Tutorial information on scone, 2018. URL `https://fahlman-knowledge-nuggets.quora.com/Tutorial-Information-on-Scone`. 1.2, 1.2

[6] Scott E. Fahlman. Mini-nuggets: Knowledge base requirements for human-like thought, 2018. URL `https://fahlman-knowledge-nuggets.quora.com/Mini-Nuggets-Knowledge-Base-Requirements-for-Human-Like-Thought`. 1.2

[7] Scott E. Fahlman. Natural language: It's all about meaning, 2018. URL `https://fahlman-knowledge-nuggets.quora.com/Natural-Language-It-s-All-About-Meaning`. 1.1

[8] Adele Goldberg. *Constructions at Work: The Nature of Generalization in Language*. New York: Oxford University Press, 2006. 1.3

[9] Rodney D. Huddleston and Geoffrey K. Pullum. *The Cambridge Grammar of the English Language*. Cambridge University Press, April 2002. 4.4.4

[10] Paul Kay. What is construction grammar? URL `http://www1.icsi.berkeley.edu/~kay/bcg/cg_define.html`. 1.3

[11] Luc Steels. *Design Patterns in Fluid Construction Grammar*. John Benjamins, 2011. 1.3

[12] Graeme Trousdale Thomas Hoffmann. *The Oxford Handbook of Construction Grammar*. New York: Oxford University Press, 2006. 1.3

[13] William A. Woods. What's in a link: Foundations for semantic networks. *Representation*

and Understanding: Studies in Cognitive Science, 1975. 1.2