**Scone Construction Grammar Engine**

**Documentation**

Yang Yang

Advisor: Professor Scott E. Fahlman

Carnegie Mellon University

Table of Contents

[Overview 3](#_Toc81858476)

[Files Included 3](#_Toc81858477)

[Overall Workflow 4](#_Toc81858478)

[Detailed Description and Algorithms 5](#_Toc81858479)

[Context 5](#_Toc81858480)

[Scone context node 5](#_Toc81858481)

[Referral context 5](#_Toc81858482)

[Constructions 5](#_Toc81858483)

[Singular nouns 7](#_Toc81858484)

[Plural nouns 7](#_Toc81858485)

[Referral nouns 8](#_Toc81858486)

[Parallel structure 8](#_Toc81858487)

[Possessive noun phrase 9](#_Toc81858488)

[Adjectives 10](#_Toc81858489)

[State verb static description 11](#_Toc81858490)

[Type-role static description 11](#_Toc81858491)

[Have has relation 12](#_Toc81858492)

[Modifiers and prepositional phrase 12](#_Toc81858493)

[Matcher and Constructor 13](#_Toc81858494)

[Matching names 16](#_Toc81858495)

[Matching pronouns 16](#_Toc81858496)

[Morphology 17](#_Toc81858497)

[Tenses 17](#_Toc81858498)

[Core NLU 18](#_Toc81858499)

Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.**Error! Hyperlink reference not valid.

# Overview

Scone Construction Grammar Engine is a Scone based systems designed for the study of the application of Construction Grammar in Natural Language Understanding. The details about how to use the engine is included in the User Manual.

This documentation includes more implementation detail of the CxG engine, and the algorithms used in the system.

# Files Included

**dictionary.lisp**

This script defines new Scone elements that is used for testing the Scone Construction Grammar Engine.

**grammar.lisp**

This script defines construction grammars and macro for creating new constructions.

**matcher.lisp**

This script implements the matcher and the constructor. The matcher is responsible for matching raw text with the construction patterns. The constructor takes in raw text, applies matched construction actions, and collects the results.

**engine.lisp**

This script implements the core nlu engine, which supports reading a sequence of raw text, back tracking and updating knowledge based on the input texts.

**cxgEngine-loader.lisp**

Loader for Scone and the Construction grammar engine.

**test.lisp**

The script includes some tests for the Scone Construction grammar engine.

# Overall Workflow

Raw Text new construction

(Referral) Context Matcher Constructions

backtrack

previous results

Previous text and Constructor

Previous used context If null

update

Text Reader

Unused outcomes

One of the possible outcomes

The whole construction grammar system consists of five major parts: Constructions storage, Context, Matcher, Constructor, text and context storage and the Text Reader.

Construction storage is responsible for storing the new constructions defined by the user.

Context has two parts: scone context node and a referral context.

Referral Context saves the current referral context the system is using right now. Since in natural language, people often refer to individuals they mentioned in previous text, the referral context serves the purpose of storing those individuals.

Construction grammar is the study of linguistic pattern and meaning. Matcher is used for matching raw natural language text with the construction patterns based on the current using referral context.

When matchers match the text with a pattern and outputs the variable values, the constructor will apply the construction actions and collect every possible outcome.

Text Reader is the core system for managing results and collecting the previously input texts.

When the Constructor gives at least one result, the system takes one of the results to update the referral context and save all the unused results in the Previous text and context storage.

If the Constructor gives a null result, that means the new input text does not have a valid meaning in the current context. The system will backtrack previously saved unused context and search for a valid one.

# Detailed Description and Algorithms

## Context

The context consists of two parts: Scone context node and referral context.

### Scone context node

Since in scone every node has a context wire, so it is important to specify the context when applying the construction rules. For example, when the system takes in “a mouse”, it will create two context nodes, where under one of the contexts, a new **{mouse}** individual will be created and under the other context, a new **{computer mouse}** individual will be created.

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.

### 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 stacks is that when people refer to an individual in previous context, they often refer to the most recently mentioned ones.

For example, starting from a NIL referral context, if the system first creates a 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})**.

## Constructions

The macro **new-construction** is defined for users to create new constructions. Detailed formatting can be found in the User Manual. When the user calls the **new-construction**, the system creates a new construction class and store it in the list **\*constructions\***.

**(defclass construction () (**

**(ret-tag**

**:initarg :ret-tag**

**:type keyword**

**:initform NIL**

**:accessor construction-tag**

**:documentation "The return tag that the construction produce.")**

**(pattern**

**:initarg :pattern**

**:type cons**

**:accessor construction-pattern**

**:documentation "The pattern of the construction,which is represented**

**as a list of components and each component is a list of alternatives.")**

**(variable-constraint**

**:initarg :var-constraint**

**:type cons**

**:accessor construction-var-constraint**

**:documentation "The list of constraints of the variables in the**

**construction method.")**

**(modifier**

**:initarg :modifier**

**:type cons**

**:initform NIL**

**:accessor construction-modifier**

**:documentation "The modifiers in the construction.")**

**(action**

**:initarg :action**

**:type function**

**:accessor construction-action**

**:documentation "The function that put the construction into Scone.")**

**(doc**

**:initarg :doc**

**:type string**

**:initform ""**

**:accessor construction-doc**

**:documentation "The documentation of the construction grammar"))**

**(:documentation**

**"A class that represent the construction grammar in the languange."))**

In the **new-construction** macro, variables are input as a list of sublists containing variable symbol and the constraints. The system will replace the variable symbols in the pattern and the modifier with the index of the symbol in the list and the new pattern and modifier is saved in construction class. Then the system binds the variables with the input action and save the lambda function as the action in the construction class. Finally, the system will remove the symbols in the variable list and save only the constraints in the class.

For example:

The user creates a new construction by calling the MACRO

**(new-construction**

**:variables ((?x :adj) (?y :noun))**

**:pattern (("a" "an") ?x ?y)**

**:ret-tag :noun**

**:modifier NIL**

**:action (let ((new\_node (new-indv NIL ?y)))**

**(new-is-a new\_node ?x)**

**(add-np-to-referral ?y new\_node)**

**new\_node)**

**:doc "np new individual with adj")**

A new construction class **#<CONSTRUCTION {10035DCDE3}>** will be created and saved in **\*constructions\***. The variables in the construction class are

**ret-tag: :noun**

**pattern: ‘( ‘(“a” “an”) 0 1)**

**variable-constraint: ‘( ‘(:adj) ‘(:noun))**

**modifier: NIL**

**action: (lambda (?x ?y) (let … … new\_node))**

**doc: "np new individual with adj"**

### Singular nouns

The most common 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 construction action is intuitively **(new-indv NIL ?x)**.

For example, when the engine takes an input “an elephant”, the engine will call the function **(new-indv NIL {elephant})** and generates **{elephant 0-2819}**.

### Plural nouns

While singular forms can be easily represented as an individual node in Scone, plural forms requires a bit of extra work.

First, I assume countable nouns are tangible objects and define a new type role **{count}** for **{tangible} :(new-type-role {count} {tangible} {number})**

Then, I need to define new number nodes. For exact expressions like “two” or “a dozen of”, I create a new individual node and link this node with the number node by an eq-link. For example, **(new-indv “a dozen of” {exact number}) (new-eq {a dozen of} {12})**.

To represent inexact expressions like “many” or “some”, 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”:

**(new-indv “some” {integer range}) (x-is-the-y-of-z {2} {lower bound} {some})**

Finally, we can define the construction rule for plural form:

**(new-construction**

**:variables ((?x {number}) (?y {tangible} :noun :type))**

**:pattern (?x ?y)**

**:ret-tag :noun**

**:modifier NIL**

**:action (let ((new\_node (new-type NIL ?y)))**

**(x-is-the-y-of-z ?x {count} ?y)**

**(add-np-to-referral ?y new\_node)**

**new\_node)**

**:doc "np new individual plural")**

For example, if we have “some apples”, then the system will create a new type **{apple 0-22148}** to represent the set of apples and **{some}** is the **{count}** of **{apple 0-22148}**.

### Referral nouns

Another type of basic noun phrase is referral nouns, with the pattern **((“the”) ?x)** where ?x is a type noun element. For example, when we say “the elephant”, that means there should be a previously mentioned elephant in previous context and the phrase is referring to that elephant.

If the current referral context is **‘({elephant 0-3141})**, then taking “the elephant” as input, the engine should return **{elephant 0-3141}**.

### Parallel structure

Parallel structure is very commonly used in natural language. For example, “apple, banana and grape” should be represented as **(list {apple} {banana} {grape})**.

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 link.

However, there are also cases the variable itself should be a list. For example, when we say “A, B, C, … and D are teammates”, 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** is used in variable constraint to specify that the variable should be treated as a list in the construction action.

**(new-construction**

**(new-construction**

**:variables ((?x :noun) (?y :noun :list))**

**:pattern (?x (",") ?y)**

**:ret-tag :noun**

**:modifier NIL**

**:action (append (list ?x) ?y)**

**:doc "noun parallel structure")**

**:variables ((?x :noun) (?y :noun))**

**:pattern (?x ("and") ?y)**

**:ret-tag :noun**

**:modifier NIL**

**:action (list ?x ?y)**

**:doc "noun parallel structure")**

The above two constructions specify the construction of parallel noun structure. When the constructor takes in for example “A, B and C”, the left construction will give **(list B C)** then it will match the pattern of the right construction, and output **(list A B C)**.

Example of treating variable as a list:

**(new-construction**

**:variables ((?x {person} :list) (?y {friend of} :relation))**

**:pattern (?x ("are") ?y)**

**:ret-tag :relation**

**:modifier NIL**

**:action (let ((len (length ?x)))**

**(if (< len 2) (error 'grammar-error**

**:message "not enough agent to support the relation"))**

**(loop for i from 0 to (- len 2)**

**append (loop for j from (+ i 1) to (- len 1)**

**collect (new-statement (nth i ?x) ?y (nth j ?x)))))**

**:doc "state verb relation friend")**

This construction specifies that **?x** has to be a list, since we need at least two people so that we can say they are friends. And the action is defined accordingly.

### Possessive noun phrase

Possessive noun phrases often fall into the pattern of (?x ?y) where ?x is a possessive name or pronoun and ?y is a type-role noun. Moreover, there are two types of possessive noun phrases:

The first case is that the phrase refers to something that’s previously mentioned. For example, if the engine previously got an input “Yang and Wesley are friends”, then the engine should understand “Yang’s friend” is referring to Wesley. Therefore, what the engine do is that it loops over all **np-ele** in referral context, and checks whether it meets two conditions: **(simple-is-x-a-y? np-ele (parent-element ?y)), (find np-ele (list-all-x-of-y ?y ?x) :test #'simple-is-x-eq-y?)**. In the above example, the first condition checks that **{Wesley}** is a **{person}** (which is **parent-element {Yang}**). The second condition checks that **{Wesley}** is a **{friend}** of **{Yang}**. If there are no previously referred element satisfies both conditions, the engine will consider the second case.

The second case is that the phrase refers to a specific individual that is not previously mentioned. For example, when we say “Mark Zuckerberg and his roommate are the founders of Facebook”, here “his roommate” is another specific individual that is the roommate of Mark Zuckerberg. Therefore, in this case, the engine will first create a new individual **(new-indv NIL (parent-element ?y))**, then **(x-is-a-y-of-z new-node ?y ?x)**, and returns the **new-node**.

### Adjectives

Adjectives are also an important part in noun phrase construction and static description construction. And I am going to use color as an example to demonstrate three common constructions for using adjectives to describe nouns.

In Scone, adjectives representation normally combines with the object. For example, “red” is represented as **{red thing}**. Additionally, when we say something is red, it means the predominant color of this thing is red. Therefore, the following lines helps to strictly defines color and red color:

**(new-type {color} {tangible})**

**(new-type {colored thing} {tangible} :english '(:no-iname :adj "colored"))**

**(new-type-role {predominant color} {colored thing} {color} :english '("color"))**

**(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})**

In this case, when we say something is a **{red thing}**, it also tells the **{predominant color}** of it is **{red}**.

The first type of construction is using adjective directly describing the noun, e.g. ( (“a” “an”) ?x ?y) where ?x is an adjective and ?y is a type noun. Basically what the construction engine would do is first applying 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, taking in “a red apple”, the construction engine will first call **(new-indv NIL {apple})** and generate **{apple 0-2333}**, then the engine will call **(new-is-a {apple 0-2333} {red thing})**.

The second type of construction is using state verb, (?x (“is” “are”) ?y), where ?x is a type noun and ?y is an adjective. For example, say taking in “apples are red”, the construction engine will simply call **(new-is-a {apple} {red thing})**.

The third type of construction is using type-role directly, for example “the predominant color of apple is red”. The code level description of type-role related construction will be in the following section.

### State verb static description

The basic pattern of static description using state verb is (?x (“is” “are”) ?y), where ?x can be any type of noun and ?y could be either an individual noun node or a type noun node. These two cases actually have different implicit meaning:

When ?y is a type node, for example “elephants are animals”, the engine should use a is-a link to represent the meaning. Specifically, the action should be **(new-is-a ?x ?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”, the engine should use a eq link to represent the meaning. Specifically, the action should be **(new-eq ?x ?y)**, where in this case **(new-eq {Clyde} {elephant 0-3141})**.

In addition, there is a specific pattern 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 second case, where the engine first apply singular noun construction, generate a new individual node and connect ?x with an eq-link. However, such construction could be inefficient in space since a new individual node is generated. A more direct action is directly call **(new-is-a ?x ?y)**. For example, when we say, “Clyde is an elephant”, the engine will apply the action **(new-is-a {Clyde} {elephant})**.

### Type-role static description

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})**

However, in natural language, sometimes people might use type-role but 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”. What the engine should do is looping **\*referral\*** and see if a previously mentioned element is an **{organization}**. The following code is how such construction is defined:

**(new-construction**

**:variables ((?x :noun) (?y :type-role))**

**:pattern (?x ("is the") ?y)**

**:ret-tag :relation**

**:modifier NIL**

**:action (let ((parent (context-element ?y)))**

**(loop for np-ele in \*referral\***

**when (handler-case (simple-is-x-a-y? np-ele parent) (t nil))**

**return (x-is-the-y-of-z ?x ?y np-ele)))**

**:doc "create the y of implicit z")**

Finally, there’s another way in natural language where type-role are commonly used: “of B” is in the agent of state verb, (?x (“are”) ?y), where ?x is a list of nouns 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”). 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 construction engine would translate into ‘**((x-is-a-y-of-z {Wesley} {friend} {Yang}) (x-is-a-y-of-z {Yang} {friend} {Wesley}))**.

### Have 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 the construction engine represent such relation is still using type-role: **(new-type-role NIL ?x ?z :n {1} :english (mapcar 'car (get-english-names ?z)))**. For example, taking an input “Clyde has four legs”, the engine will construct a new type-role **{leg 0-2813}** representing the legs of Clyde with external names “leg” and “legs”.

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 also inherit the names of **{leg}**, the system will be able to understand “Clyde’s legs” in future text.

However, since the has-link functionality are 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 has-link is fully settled.

### Modifiers and prepositional phrase

When definition new constructions, one could also add modifiers to the variables. Modifiers are used to affect the variable construction, for example add a context or tense etc. Prepositional phrases are where context modifiers are used to specify the context of the describing element. For example, considering “Elephants are grey in Africa”, it matches with the construction **((?x :relation) (?y {place} :noun))**. The system would want to create the is-a link between **{elephant}** and **{grey thing}** under the context **{Africa}**. Therefore, the construction would first specify a **(new-indv nil {place})** as a modifier for **?x** and then in action use is-a link to connect the context of **?x** and **{Africa}**. The construction for location prepositional phrase is as follow:

**(new-construction**

**:variables ((?x :relation) (?y {place} :noun))**

**:pattern (?x ("in" "at" "on") ?y)**

**:ret-tag :relation**

**:modifier ((?x (new-indv nil {place})))**

**:action (progn**

**(new-is-a (context-element ?x) ?y)**

**?x)**

**:doc "location prepositional phrase")**

Similarly, one could also define constructions for other types of prepositional phrase like time etc.

## Matcher and Constructor

The Matcher takes in a raw text and a construction, tries to match the text with the pattern of the construction and returns the values of the variables in the construction.

The Constructor is responsible for iterating over all constructions and use the Matcher to match every construction with the text and apply the construction action. The Constructor collects every possible result, tag and the after-construction context.

First, the system will tokenize the text into word list. Currently the system just uses a naïve tokenizer that split the text by space and treat comma as a separate token.

Then, we can define a **varible-match** and a **one-ele-match** function. Below is the pseudo code:

**defun variable-match (text constraints):**

**extract syntax\_tag from constraints**

**l1 = for (element, \_) in (lookup-definitions text syntax\_tag):**

**when (element satisfies constraints)**

**collect (element, (copy \*referral\*))**

**l2 = for (element, \_ , context) in (constructor text syntax\_tag):**

**when (element satisfies constraints)**

**collect (element, context)**

**return (append l1 l2)**

**defun one-ele-match (text single-pattern var\_constraint):**

**case (type of** **single-pattern):**

**integer:**

**return (variable-match text var\_constraint[single-pattern])**

**string list:**

**if (find text single-pattern):**

**return text**

**else:**

**return NIL**

**variable-match** takes in raw text and a list of variable constraints. The function returns all possible pairs of Scone element value for the variable and the corresponding referral context.

In **variable-match**, **l1** is evaluated directly through looking up the text in Scone and **l2** is evaluated using the constructor (discuss latter).

**one-ele-match** takes in raw text, a single component in the pattern and the variable constraints of a particular construction. If the pattern component is a number, which means a variable, it calls the **variable-match** function to check if the variable can match the pattern. If the pattern component is a list of string, the function checks if the input text has an exact match with the string in the list.

Now, we can use **one-ele-match** to recursively define the **Matcher**. Below is the pseudo code:

**defun matcher (wordlist patternlist var\_constraint):**

**if (null wordlist) or (null patternlist):**

**return (list null null … null \*context\* \*referral\*)**

**[**the number null is equal to the number of variables**]**

**before-context = (\*context\*, (copy \*referral\*))**

**result = []**

**for i from 1 to (len(wordlist) – len(patternlist) + 1):**

**text = (join-by-space wordlist[:i])**

**first\_result = (one-ele-match text patternlist[0] var\_constraint)**

**if (type-of first\_result is String):**

**result.append(matcher wordlist[i:] patternlist[1:] var\_constraint)**

**else:**

**for (ele, ctx) in first\_result:**

**(\*context\*, \*referral\*) = ctx**

**result.append(loop for rest\_ele in**

**(matcher wordlist[i:] patternlist[1:] var\_constraint)**

**do (setf rest\_ele[patternlist[0]] ele)**

**collect rest\_ele)**

**(\*context\*, \*referral\*) = before-context**

**return result**

Note when we try to match a pattern with a word list, the first component of the pattern should be matched with a “prefix sublist”(sublist starting from index 0) of the word list. Therefore, firstly the matcher will loop every possible “prefix sublist”. Then, for every prefix sublist, the function applies one-ele-match to the joined text, change the referral context and then recursively apply matcher to the rest of wordlist and rest of the pattern. After the function collects every combination of result, it will set referral context back to what it is at the beginning of the function.

The outcome of the matcher is a list of possible results where each result is a list of values of the variable in the construction and a referral context after getting these values.

With the **Matcher** well defined, the **Constructor** will apply the matcher on every construction. The pseudo code of the **Constructor** is as below:

**defun constructor (text &optional taglist):**

**before-context = (\*context\*, (copy \*referral\*))**

**result = []**

**for construction in \*constructions\*:**

**extract pattern, constraint, tag, action from construction**

**if (not null taglist) and (tag not in taglist): continue**

**\*context\*, \*referral\* = before-context**

**matcher\_results = (matcher (tokenizer text) pattern constraint)**

**if (not null matcher\_results):**

**result.append(for match\_result in matcher\_results:**

**extract variable\_value, context from match\_result**

**\*context\*, \*referral\* = context**

**collect (apply action variable\_value)**

**\*context\*, \*referral\* = before-context**

**return result**

For every result the system gets from the matcher, the constructor will set the referral context to the context from the matcher result, then apply the construction action and finally collect the action element and the after-action referral context. After the function collects every possible construction result, it will set referral context back to what it is at the beginning of the function.

An example of how the matcher and constructor work:

Say we have an input text **“****an elephant kicks a mouse”**,

Constructor will try to match the text with every construction,

Consider the construction :

**(new-construction**

**:variables ((?x {animal} :noun)**

**(?y {kick} :verb)**

**(?z {physical object} :noun))**

**:pattern (?x ?y ?z)**

**:ret-tag :verb**

**:action (let ((new\_v (new-indv NIL ?y)))**

**(x-is-the-y-of-z ?x {action agent} new\_v)**

**(x-is-the-y-of-z ?z {action object} new\_v)**

**new\_v)**

**:doc "transitive action kick")**

Since **(constructor “an elephant” ‘(:noun))** returns **{elephant 0-3142}**, the matcher will match “an elephant” with the first component of the pattern. Then the constructor will recursively call the matcher on **“kicks a mouse”** and **(?y ?z)**.

Similarly, “kicks” matches with {kick} and **(constructor “a mouse” ‘(:noun))** returns **{mouse 0-2816} and {computer mouse 0-2818}**, which are both **{physical object}**.

Therefore, the matcher will return **({elephant 0-3142} {mouse 0-2816}) and ({elephant 0-3142} {computer mouse 0-2818})** together with the corresponding referral context.

Then, the constructor will apply the action on **({elephant 0-3142} {mouse 0-2816}), ({elephant 0-3142} {computer mouse 0-2818})** and gets **({kick 0-2824} context1 referral1) and ({kick 0-2830} context2 referral2)**.

### Matching names

Name matching is tricky since first it is hard to define all possible names in advance (there are too many names) and second in natural language, people sometimes assume the name represents a person or a place, etc.

To resolve the first problem, in the function **variable-match (text constraints)**, when the text does not have any existing matched element, the system will detect if the text is a name. The criterion is 1. The first letter of every word in text needs to be uppercase 2. If there’s syntax tag constraint, it needs to be :noun 3. The constraints cannot have :type or :list. If the conditions are matched, the system will create a new individual node with text as its iname.

To resolve the second problem, some adjustments are made when the system tries to determine if an element meets a constraint. If the iname of the element and the constraint satisfy the above conditions, the system extract all the parent nodes in the constraint and create is-a links between every parent and the element.

Note, since both above two cases are still guessing and proceed, the system will create a new context.

Example: If the matcher tries to match “Yang and Wesley are friends” with the construction

**(new-construction**

**:variables ((?x {person} :list) (?y {friend of} :relation))**

**:pattern (?x ("are") ?y)**

**…)**

When constructing “Yang and Wesley”, the system will create **(new-indv {Yang} {thing})** and **(new-indv {Wesley} {thing})**. Then since **?x** is constrained to be **{person}**, the system will create a new context node **(in-context (new-context NIL \*context\*))**, and create two is-a link **(new-is-a {Yang} {person}) (new-is-a {Wesley} {person})**.

### Matching pronouns

The most common pronouns people use in natural language are “he”, “she”, “it”, “they” and the corresponding objective pronouns and possessive pronouns.

“he”, “him” and “his”:

Since “he” normally refers to a previously referred male, the system needs to look into the referral context and get 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 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” and “her”:

This is like “he” and “him” except that the pronoun refers to a **{female person}**.

“it” and “its”:

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” and “their”:

“they” 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 **\*referral\***.

### Morphology

Morphology is also an essential part of the construction, and it also contains underlying information. There are lots of related research about morphology which is not the focus of this system. Currently I am just providing a very simple morphology function which takes in a word and output the root word and morphology tags. The user is subject to change this morphology function with any better ones.

While defining constructions, one can also use variables to collect the morphology tags. For example, the text matches the pattern **(?x “is”)** when the root of the text is “is” and the morphology tags will be saved in **?x**. For instance, “are” will match the pattern and the corresponding **?x** will be **:plural**; “was” will also match the pattern and **?x** will be **:past**.

### Tenses

In English, tenses are expressed through the tense of the verb, which could be acquired through getting the morphology of the verb. Take past tense for state verb as an example:

Firstly, we need to create a **{time interval}** representing the past:

**(new-indv {past} {time interval})**

**(x-is-the-y-of-z {now} {time interval finish} {past})**

Secondly, in the construction, instead of matching the verb string directly, we would want to collect the morphology tags of the verb. For state verb, the variable would be **(?v "is")**.

Finally, in the action of the construction, when **?v** has **:past**, the remaining actions of the construction should be in the **{past}** context.

Given the above three steps, we could modifier the “state verb adj” constructions to the following that takes tense into consideration:

**(new-construction**

**:variables ((?x :noun) (?v "is") (?y :adj))**

**:pattern (?x ?v ?y)**

**:ret-tag :relation**

**:modifier NIL**

**:action (progn**

**(if (find :past (cdr ?v)) (in-context (new-indv nil {past})))**

**(if (find :future (cdr ?v)) (in-context (new-indv nil {future})))**

**(add-np-to-referral ?x)**

**(new-is-a ?x ?y))**

**:doc "state verb adj")**

## Core NLU

The purpose of the core NLU engine is to make the system able to understand a sequence of texts using construction grammar.

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 different referral context, and different referral context would give different understanding of future texts, the understanding of a sequence of texts is like a tree structure:

Text Reader Original Context

Text 1 meaning 1 meaning 2 meaning 3 …

Context 1 Context 2 Context 3

meaning 11 meaning 12 meaning 31 meaning 32 …

Text 2

Context 11 Context 12 Context 21 Context 22 Context 31 Context 32

Text 3

…… ……. ……. …… ……

We can see the tree could grow extremely large when the Text Reader read more and more texts. Therefore, I decided not to collect every possible result but take one result at each time to continue and save all the other results for future look back if the chosen one does not make sense for future texts.

Text Reader Original Context

Text 1 meaning 1 meaning 2 meaning 3

(save Context 2, Context 3) Context 1 Context 2 Context 3

Text 2 meaning 11 meaning 12

(save Context 21) Context 11 Context 21

Text 3

… … ….

Pseudo code for implementing the above algorithm:

**\*text-record\* is used for saving the current using track**

**\*result-record\* is a list of unused referral context lists**

**defun checker (context textlist):**

**before-context = (\*context\*, (copy \*referral\*))**

**\*context\*, \*referral\* = context**

**result = (iterate over all possible meaning and referral context path of the texts in the textlist, if one of the paths gives not null meaning for every text, returns T otherwise returns NIL)**

**\*context\*, \*referral\* = before-context**

**return result**

**defun backtrack (unread\_text):**

**if (checker \*context\* \*referral\* unread\_text): return unread\_text**

**if ((last \*result-record\*) is null list):**

**if (null \*text-record\*): return nil**

**new\_unread = (append (last saved text in \*text-record\*) unread\_text)**

**remove the last \*text-record\***

**remove the last \*result-record\***

**return (backtrack new\_unread)**

**else:**

**extract the first (text, meaning, ctx) from last \*result-record\***

**removed the extracted info from \*result-record\***

**if (checker ctx unread\_text):**

**\*context\*, \*referral\* = ct**

**save text and meaning into \*text-record\***

**return unread\_text**

**return (backtrack unread\_text)**

**defun read-text (text):**

**construction\_result = (constructor text)**

**if (not null construction\_result):**

**extract ele, tag, ctx from the first construction\_result**

**store (text, ele, tag) in \*text-record\***

**store the rest of construction\_result in \*result-record\***

**\*context\*, \*referral\* = ctx**

**return \*text-record\***

**else:**

**unread = (backtrack ‘(text))**

**if (null unread): return NIL**

**for new\_text in unread:**

**(read-text new\_text)**

**return \*text-record\***

If in current context, constructor gives non null result, it means the input text make sense in the current context. Then the system will take the first result and update \*text-record\* and \*result-record\*.

If the constructor gives null result, that means the text does not make sense in current context, so we need to backtrack \*result-record\*. The function backtrack takes in a list of unread text then locate the closest state in \*result-record\* where all the future texts make sense and returns the list of unread text from the located state. Then the system will call the read-text function on all the unread texts.

Detailed usage and example of **read-text** function is included in the User Manual.

Details about verbose mode is also included in the User Manual.