**Scone Construction Grammar Engine**

**Documentation**

Yang Yang

Advisor: Professor Scott E. Fahlman

Carnegie Mellon University

Table of Contents

[Overview 3](#_Toc76683187)

[Files Included 3](#_Toc76683188)

[Overall Workflow 4](#_Toc76683189)

[Detailed Description and Algorithms 5](#_Toc76683190)

[Constructions 5](#_Toc76683191)

[Plural nouns 6](#_Toc76683192)

[Parallel structure 7](#_Toc76683193)

[Context 8](#_Toc76683194)

[Scone context node 8](#_Toc76683195)

[Referral context 8](#_Toc76683196)

[Matcher and Constructor 9](#_Toc76683197)

[Matching names 11](#_Toc76683198)

[Core NLU 12](#_Toc76683199)

# 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

## 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.")**

**(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 with the index of the symbol in the list and the new pattern 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**

**: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))**

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

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

### Plural nouns

While singular forms like ((“a” “an”) :noun) 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}**.

### 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 needs to be a list and should be treated as a list in the construction action.

**(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")**

**(new-construction**

**: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.

## 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 association list in the system. The keys are type nodes, and the datum are stacks of individual nodes whose parents are the keys. 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, the system first creates a new **{elephant}** individual **{elephant 0-3141}**. Then, the referral context will be **‘(‘({elephant} . {elephant 0-3141}))**. Now, the system creates another **{elephant}** individual **{elephant 0-3142}**, then the referral context will be **‘(‘({elephant} . {elephant 0-3142} {elephant 0-3141}))**. In this case, if the system wants to extract a previously referred **{elephant}**, it will first get **{elephant 0-3142}**.

## 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})**.

## 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 Referral Context: NIL

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 Referral Context: NIL

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\* = ctx**

**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.