**Scone Construction Grammar Engine**

**User Manual**

Yang Yang

Advisor: Professor Scott E. Fahlman

Carnegie Mellon University

Table of Contents

[Overview 3](#_Toc76655746)

[Running Scone CxG Engine 3](#_Toc76655747)

[Adding New Constructions 3](#_Toc76655748)

[Construction Matcher and Constructor 6](#_Toc76655749)

[Core NLU Engine 7](#_Toc76655750)

[Verbose Mode 8](#_Toc76655751)

# Overview

Construction grammar is a field of linguistics study of learning the pair of the linguistic patterns and the underlying meaning. The study of construction grammar can be used for Natural Language Understanding AI system for the purpose of learning the actual meaning of language.

Scone is a knowledge-based system comprised of a search/inference engine and a knowledge base of common knowledge. The meaning of language can be saved in Scone and the Scone Construction grammar engine is designed for the purpose of studying Natural Language Understanding.

Scone Construction grammar engine provides an interface for the user to easily create new constructions. Additionally, the engine can take in raw text and use apply existing construction grammar rules to get the meaning of the text and input into the Scone knowledge base. While words and sentences could have multiple meanings, the engine will record all possible meanings throughout the input of multiple texts and when current context does not make sense, the engine will look back and search for meanings that works for the inputs.

# Running Scone CxG Engine

The user needs to put the dictionary.lisp, grammar.lisp, matcher.lisp, engine.lisp and cxgEngine-loader.lisp under the same directory of Scone. Then change the variable rootpath in cxgEngine-loader.lisp to the path of the root directory of Scone.

In our typical setup, start Steel Bank Common Lisp, then load the cxgEngine-loader.lisp file to start Scone and the CxG Engine.

# Adding New Constructions

**\*constructions\***

A list of constructions defined and saved in the system.

**\*referral\***

An association list whose keys are Scone type elements and values are the individual nodes that refer to this element in previous text.

For example, when the engine takes in “an elephant kicks a mouse”, it matches “an elephant” with **{elephant 0-2656}**. Then the system will add **(list {elephant} {elephant 0-2656})** to **\*referral\***. If the system takes in a new individual **{elephant 0-2657}**, then **\*referral\*** will be changed to **(list {elephant} {elephant 0-2657} {elephant 0-2656})**.

**new-construction (&key variables pattern ret-tag action doc) [MACRO]**

This macro is used for defining new construction. It takes in variables representing the Scone elements in the pattern, the pattern of this construction, the return tag of this construction, the action of inputting the construction implied meaning to Scone knowledge base and the documentation of this construction. Finally, the macro adds the new defined construction to **\*constructions\***.

**variables** is in the format of a list of list that represents each variable. The first element of the list is the symbol used for the variable, conventionally a question mark and a letter, i.e.?x, ?y, etc. The rest of the list are the constraints for the variable:

|  |  |  |
| --- | --- | --- |
| Constraint | Example | Explaination |
| Scone element | {animal} | The variable needs to be an animal |
| Syntax tag | :noun | The variable is a noun |
| Type/indv | :indv | The variable is an individual |
| Is a list | :list | The variable is a list of results |

For example, **(?x {animal} :noun :type)** means the variable is represented as **?x** in pattern and action, it is a **{animal}** in Scone, its syntax tag is :noun and it is a type node.

**pattern** is in the format of a list of components where each component can be either a variable or a list of string alternatives.

For example, the pattern **((“a” “an”) ?x)** represents the first component of the pattern can be either “a” or “an” and the second component of the pattern should match the variable **?x**.

**ret-tag** is the syntax tag of the Scone element output by the **action**.

For example, for noun phrase constructions, the **ret-tag** would be **:noun**.

**action** is the lisp code for putting the meaning bond with the construction into Scone. The action is binding with the local variables defined in **variables**. The action can also manipulate **\*referral\*** if necessary.

Here are a couple of examples in grammar.lisp to better understand the use of this macro:

**(new-construction**

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

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

**:ret-tag :noun**

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

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

**new\_node)**

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

Note the function **add-np-to-referral** is a helper function that add the element to **\*referral\***. This construction binds the local noun variable **?x** and the construction action first add the new individual to **\*referral\*** and then return this new individual node.

**(new-construction**

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

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

**:ret-tag :noun**

**:action (let ((try-find (assoc ?x \*referral\* :test #'simple-is-x-eq-y?)))**

**(if (not (null try-find)) (car (cdr try-find))**

**(error 'grammar-error :message "cannot find the referral noun")))**

**:doc "np referral individual")**

In natural language, for example when we say, “the elephant”, that means we already know an elephant in current context. The action of this construction will try to find **{elephant}** in **\*referral\*** and return the most recent referred individual if it exists.

**(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")**

This construction defines a new construction for transitive action kick. Since only an animal could perform the action kick, the construction constraint the variable **?x** to **{animal}**. Similarly, only a physical object can be kicked, the construction also bind **?y** to **{physical object}**.

**(new-construction**

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

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

**:ret-tag :relation**

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

**:doc "state verb type")**

**(new-construction**

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

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

**:ret-tag :relation**

**:action (new-eq ?x ?y)**

**:doc "state verb indv")**

When we say, “Clyde is an elephant”, it means **{Clyde}** is equivalent with the new elephant individua. However, when we say, “elephants are animals”, that means **{elephant}** is a **{animal}**. The variable constraints **:type :indv** help to distinguish the two cases.

**(new-construction**

**:variables ((?x {person} :list) (?y {teammate 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 teammate")**

The above construction is an example of using keyword :list. When the system applies construction rules to “A and B”, it would return **(list A B)**. Therefore the above construction patterns could match with “A and B are teammates” or “A, B and C are teammates”.

# Construction Matcher and Constructor

**constructor (text &optional taglist verbose) [FUNCTION]**

The constructor function takes in a raw text and tries to match with all existing constructions. If the text match with the pattern of the construction, the system applies the action and collect the result. The function returns a list of 4-element list, where each 4-element list corresponds to a possible outcome. The first element is the output Scone element, the second element is the return syntax tag, the third element is after construction scone context node and the final element is the after-construction referral context. Return NIL if no construction matches. If the function takes in a list of syntax tags, the system will only check constructions that has the return tag in the list.

When verbose is true, the construction will be in a verbose mode (discuss later).

For example, when **\*referral\*** is NIL,

Text

Description automatically generated

# Core NLU Engine

**\*text-record\***

This saves all previously input raw text and current interpretation of the text. The record is saved as a list of 3-element lists in order, where the first element is the raw text, the second element is the Scone element for the text and the third element is the syntax tag.

**\*result-record\***

This saves all possible outcomes of the constructor for every previous input text that are not used.

For example, when the system takes the input “an elephant kicks a mouse”, then constructor could get two possible outcomes (showed in the example above). The system will naively take the first result, set **\*referral\*** as **(({mouse} {mouse 0-2896}) ({elephant} {elephant 0-2881}))**, set **\*context\*** as **{general 0-2895}**, save **{kick 0-2907}** in **\*text-record\*** and save **({kick 0-2913} :VERB {general 0-2898} (({computer mouse} {computer mouse 0-2899}) ({elephant} {elephant 0-2881})))** in **\*result-record\***. **\*result-record\*** is used for the purpose of backtracking if we realized current referral is incorrect when the system got more text.

**read-text (text verbose) [FUNCTION]**

This function takes a text in natural language (English) and applies **constructor** to the text. If the constructor get non-nil results, the system naively takes the first result, set new **\*referral\***, set new **\*context\***, save the result in **\*text-record\***, and save the not used results in **\*result-record\***. If the constructor returns nil, then the system will look back to previously unused referral context and reread the texts. The function returns the **\*text-record\*** after reading the newly input text.

When verbose is true, the engine will be in a verbose mode (discuss later).

For example,

Text

Description automatically generated

We can see here, initially, when reading “Clyde kicks a mouse”, the system takes mouse as animal mouse. However, when reading “Yang made the mouse”, the system will realize mouse is a **{man-made object}** and look back into **\*result-record\***. Therefore, after reading the last sentence, the system will take mouse as **{computer mouse}** and change the meaning of the second sentence as well.

# Verbose Mode

The user can use the verbose mode of the NLU engine by calling the function **(read-text text t)**.

The verbose mode of the constructor can used by calling the function **(constructor text NIL t)**.

In verbose mode, the constructor will print out the message every time a piece of text correctly matched with a piece of text. Also, when the construction will print out the assumptions it made with names. The core NLU engine will print out messages when it’s looking back into previous record.

Example:

Text

Description automatically generated

When we load the above test script, we have:

Text

Description automatically generated