A Prototype of a Natural Language Toolkit in Clojure

J. Angell
School of Computing
Teesside University
M2012929@tees.ac.uk

M. Cataldo
School of Computing
Teesside University
G7130816@tees.ac.uk

M. Chamberlain
School of Computing
Teesside University
M2014976@tees.ac.uk

Keywords: Natural Language Toolkit, COM3036-N-BJ1-2015

Abstract

To engineer a prototype version of a natural language toolkit forming the foundations of a system like LKit (Lynch 2016) / NLTK within Clojure.

1 Background

Natural language processing (NLP) is used to interpret spoken language both syntactically and semantically. To achieve reliable results an input, that is to be parsed, needs to undergo several stages of processing, collectively referred to as a natural language toolkit. (NLTK 2016) is the most accessible example of such a toolkit and is a free, open source and community driven project written in Python with capabilities for tokenizing, parsing and named-entity recognition.

NLP involves several distinct stages, the first of which being morphological processing. This preliminary stage is used to tokenize sentences to form a collection of discrete words. For instance, some words may be split into several sub-words and punctuations such that the word 'unhappily' becomes three new tokens 'un', 'happy' and 'ly', this process is known as inflection. (Manning 1999)

Morphological processing is then followed by the syntax analysis stage which is concerned with de-constructing a sentence into discrete syntactic forms using a specialised lexicon and grammar. The lexicon serves as a dictionary of word definitions that a syntactic analyser (or parser) must use to lookup a word's syntactic category and other related semantic information. Whereas, the grammar is a formal definition of the language being parsed, defining a set of syntax rules. These rules can be used to categorise components of a sentence in order to show their relationships within a sentence. As part of the syntax analysis stage, a top down parser may be used to achieve a naive or predictive approach in the sense that the algorithm works from a completed sentence and attempts to parse down into its composite grammatical forms. This is in contrast to a bottom up approach, where the algorithm initially tokenises a sentence into its word components in order to form a tree-like structure that replicates the semantic meaning of the original sentence.

Often NLP is achieved using statistical models. Arguably the main statistical model used within statistical natural language recognition systems is the Hidden Markov Model (Manning 1999).

2 Introduction

This paper will explore the development of a prototype version of a natural language toolkit written in Clojure and features of the language will be discussed. Specifically, this paper will highlight some of the main language constructs used in this development to achieve tasks that would typically require further development work in similar languages (Lisp).

Furthermore, this paper will explore the differences in common parsing algorithms and explain some of the decisions taken during development. The paper will mainly focus on the syntactic analysis stage of the toolkit with the incorporation of a specialised lexicon and grammar. The syntactic analysis stage will be accomplished by a chart parser. The toolkit is designed to support many parsing mechanisms, as commonly seen in existing toolkits such as (NLTK 2016), however the initial implementation focuses on a chart parsing mechanism. The paper will evaluate the parser using a collection of tailored sentences to validate behaviour and identify potential limitations. One such limitation could arise from structural and word sense ambiguity which was considered during the design.

It was also important to consider the results yielded by various parsing techniques and examine how they may differ. Differences in parsing algorithm may give rise to execution time issues, omission of valid parses and varying degrees of edge storage. The chart parser that forms the basis of this work uses a bottom up approach, such that all intermediary edges are stored leading up to a complete parse.

A statistical approach was not chosen as semantics will not be the main focus of the work. The parser discussed in this paper is able to be extended to incorporate more intricate semantic processing.

3 Lexicon

The lexicon is defined as a Clojure map data structure where every word is stored uniquely as a key, this enables efficient lookup of word definitions. Each entry represents a word's lexical definition which includes syntactic category information and its semantic meaning. This information is also stored using a map data structure as shown Figure 1.

During development a problem was faced regarding ambiguous words with more than one semantic meaning. For example the word 'Duck' can reside in both the verb and noun syntactic categories. This was addressed by supporting a list of word definition entries for a single lexicon key as shown in Figure 2. A type check could then be performed and each syntactic categories information could be iterated over in order to explore an edge for each word definition.

```
:dog { :syncat 'noun
       :sems  'canine
       :tags  '#{animate animal}
       :num  'singular
     }
```

Figure 1: Example of a lexicon entry

```
:duck '({ :syncat noun
          :sems   (animal duck)
          :tags   #{animate animal}
          :num    singular
        }
        { :syncat verb
          :sems   (crouch)
          :tags   #{activity crouch}
          :num    singular
        })
```

Figure 2: Example of a lexicon entry with multiple meanings

4 Grammar

The grammar is a formal definition for the structure of a language. A context free-grammar has been used in that no context information is taken into account when forming parse trees. Both context free-grammars and context-sensitive grammars are limited in the sense that they cannot fully describe all languages, but context-sensitive grammars are more generic and therefore are more descriptive. The scope of the current work however is concerned with the actual parsing engine therefore the complexity of a context-sensitive grammar adds is unnecessary.

```
(S [sentence -> noun-phrase verb-phrase]
   { :actor NP
     :action (:act VP)
     :object (:obj VP)
   })
```

Figure 3: Example of a grammar entry

Figure 3 shows the grammar rule S in Clojure. The rule outlines that the two consequent syntactic categories noun-phrase and verb-phrase can be compounded to achieve a construct with the syntactic category sentence. The rule is classified by an assigned name which uniquely identifies each rule within the grammar. This is followed by the antecedent that represents the resulting grammatical structure upon fulfilling the rule's consequents. Also represented are rules for semantic slot filling that aggregate consequent meaning to form an overall definition for the antecedent, in this case the sentence.

A grammar rule is recursive if a syntactic category is present in both the antecedent and consequent, for example the rule $S \longrightarrow S\ PP$ involves direct recursion with S.

5 Chart Parser

The chart parsing procedure takes a sentence as input and aims to exhaustively explore its meaning. Success is highly dependent on the quality of the predefined grammar on which the parser operates. The parsing algorithm specifically processes the input from left to right, sequentially filling a chart with map structures classified as edges. The chart is of size $L + 1$, where L is the length of the input. An edge represents a completed or partial parse, spanning a group of words. A completed edge which spans L is considered a successful parse [TODO ADD REF].

Structural ambiguity occurs when the grammar assigns more than one possible parse tree to a sentence. Determining which sentence is the most appropriate may be impossible without additional semantic information. As mentioned previously, the parser adopts a bottom up approach which starts with individual elements of the input rather than the complete sentence. This approach does not make assumptions about the input such that potential parses would be overlooked.

```
{ :consumed-edges nil,
  :rule word,
  :start 1,
  :semdef {
          :sems canine,
          :tags #{animate animal},
          :num singular,
          :root :dog},
  :targets nil,
  :components ({
          :syncat noun,
          :semdef {
                  :sems canine,
                  :tags #{animate animal},
                  :num singular,
                  :root :dog}}),
  :syncat noun,
  :status :complete,
  :end 2}
```

Figure 4: Terminal node chart edge

Figure 4 shows an edge which is taken from a complete parse for the phrase ‘The dog chased the cat’ spanning the word ‘dog’. The semantic information for the edge has been acquired using the lexicon and its rule has been derived from the grammar. The value of consumed-edges stores a tree structure of all extended edges that have been consumed to form this edge. An edge’s status can be either `:complete` or `:active` where an edge being active means it is incomplete and has existing targets. A edge’s targets contains a collection of syntactic categories that the edge is actively seeking in order to become complete.

```
foreach word in sentence
        Lookup word in lexicon
        Create word edge
        Add complete edge to the chart for a word
        Add relevant active edges from grammar
        Extend current active edges
```

Figure 5: Parser Pseudo

Figure 5 outlines the parsing process used to produce an exhaustive chart. Following the lexicon lookup, a complete, terminal edge is added to the chart for each word. On adding a complete edge relevant active edges are also added to the chart. An edge is relevant if it's related grammar rule contains an entry within its consequents for the same syntactic category as the triggering edge. For example if the completed edge was for a determiner, an edge with the rule for a noun-phrase would be relevant e.g [noun-phrase $\longrightarrow$ determiner noun].

```
(defn get-grammar-rule-name [rule]
  (let [[a -> b & c] (second rule)] a))
```

Figure 6: Helper Function to obtain the antecedent of a rule

Figure 6 showcases Clojure's map destructuring feature that makes screening a rule for grammatical structures of interest trivial. It shows a helper function that returns the rule's non-terminal grammatical structure, the formation of which is detailed after the '->' (This element is included for readability purposes only). This is achieved as elements within the structure are sequentially bound to the constructs within the let form, these newly bound constructs can then be referenced within the scope of the let form, in this case one of constructs is simply returned. The ampersand is included within the let form to handle the case in which the antecedent of the grammatical rule is longer than two. This case is handled as the ampersand forms a rest seq with the remainder of the rule's elements.

A complete sentence parse is achieved when an edge spans the length of the original sentence. A fully exhausted chart may result in multiple complete sentence parses. This is dependant both on the nature of the sentence and the construction of the grammar rules themselves. For example, a successful completed parse for the the sentence 'The dog chased the cat' would yield the parse tree as shown in Figure 7.

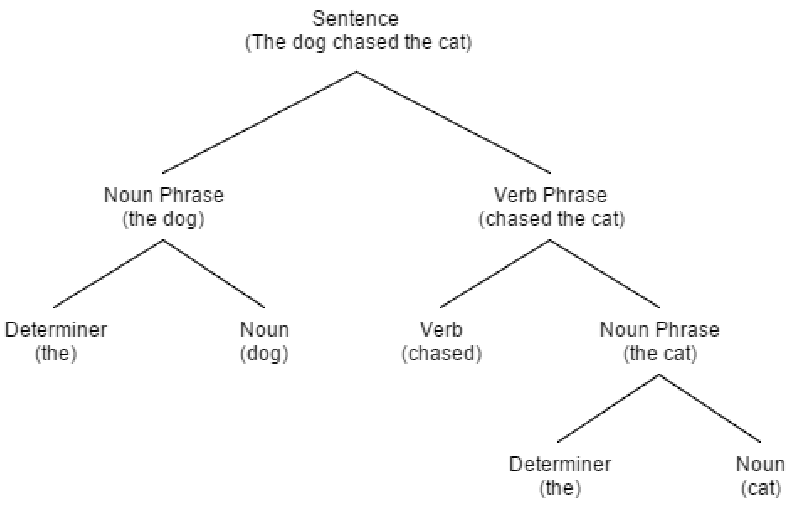

Figure 7: Parsed tree example

6 Evaluation

Clojure proved to be an effective language for the development of an NLP parser as many of the data structures within the system were maps. TODO justify / comment on why clojure and how it was good for project

A collection of sentences were comprised both to test the grammar and to evaluate the chart parser. Each sentence builds on the previous by adding further semantic meaning and grammatical structures. All potential parses are explored such that no valid meaning is lost during the process. If a complete parse is unachievable, a partial parse can be obtained from the result.

```
--> sentence (:the :dog :chased :the :cat) 0 5 () :complete
   --> noun-phrase (:the :dog) 0 2 () :complete
      --> det (:the) 0 1 nil :complete
      --> noun (:dog) 1 2 nil :complete
   --> verb-phrase (:chased :the :cat) 2 5 () :complete
      --> verb (:chased) 2 3 nil :complete
      --> noun-phrase (:the :cat) 3 5 () :complete
         --> det (:the) 3 4 nil :complete
         --> noun (:cat) 4 5 nil :complete
```

Figure 8: Completed parse for ‘The dog chased the cat’

Figure 8 shows an example of a basic sentence that satisfies the simplest grammar rule for a complete sentence, ‘The dog chased the cat’. It shows a root node with the completed status of the syntactic category type `sentence` which spans the whole length of the input. The resultant tree structure is as expected with the provided grammar rules and therefore is considered a successful parse.

```
--> sentence (:the :quick :dog :chased :the :brown :cat) 0 7 () :complete
   --> noun-phrase (:the :quick :dog) 0 3 () :complete
      --> det (:the) 0 1 nil :complete
      --> adj (:quick) 1 2 nil :complete
      --> noun (:dog) 2 3 nil :complete
   --> verb-phrase (:chased :the :brown :cat) 3 7 () :complete
      --> verb (:chased) 3 4 nil :complete
      --> noun-phrase (:the :brown :cat) 4 7 () :complete
         --> det (:the) 4 5 nil :complete
         --> adj (:brown) 5 6 nil :complete
         --> noun (:cat) 6 7 nil :complete
```

Figure 9: Completed parse for 'The quick dog chased the brown cat'

The output shown in Figure 9 illustrates a modified version of the above sentence adding adjectives to each noun. As shown the parser is able to manage the introduction of adjectives, however multiple adjectives in sequence would require a more advanced grammar whereby they are targeted recursively.

```
--> sentence (:the :quick :dog :chased :the :brown :cat :under :the ...
   --> sentence (:the :quick :dog :chased :the :brown :cat) 0 7 ()
      --> noun-phrase (:the :quick :dog) 0 3 () :complete
         --> det (:the) 0 1 nil :complete
         --> adj (:quick) 1 2 nil :complete
         --> noun (:dog) 2 3 nil :complete
      --> verb-phrase (:chased :the :brown :cat) 3 7 () :completed
         --> verb (:chased) 3 4 nil :complete
         --> noun-phrase (:the :brown :cat) 4 7 () :completed
            --> det (:the) 4 5 nil :complete
            --> adj (:brown) 5 6 nil :complete
            --> noun (:cat) 6 7 nil :complete
      --> prep-phrase (:under :the :tree) 7 10 () :complete
         --> prep (:under) 7 8 nil :complete
         --> noun-phrase (:the :tree) 8 10 () :complete
            --> det (:the) 8 9 nil :complete
            --> noun (:tree) 9 10 nil :complete
```

Figure 10: Completed parse for 'The quick dog chased the brown cat under the tree'

Figure 10 applies a prepositional phrase to the end of the sentence, adding further semantics for a location. This creates ambiguity as the phrase gives the sentence two possible meanings, although the resulting tree will be the same. This also shows the application of a recursive grammar rule as earlier mentioned.

Depending on the size of the grammar the parsing process is more expensive and past a certain size would become less feasible. Certain language complexities could lead to a parses chart becoming exceedingly large in certain cases.

7 Future Work

7.1 Morphology

Prior to applying the chart parsing procedure, morphological processing can be applied to a sentence to reduce the size of the lexicon. These processes include inflection, pluralisation and compounding. Inflection is the modification of a word to express different grammatical categories for example walk to walking.

Morphology, temporarily remove punctuation, headache to head ache, handle pluralisation (pre process before chart parsing)

7.2 Semantics

A large element of future work is involved with the semantic processing of words after they have been parsed. The current state of the development does not deal with semantic processing. Semantics are combined when two terminal grammatical structures are combined to form a non-terminal grammatical structure. The next stage in NLP would be to process this information, possibly including some case frame analysis to help resolve parse ambiguity.

Information would be placed inside a slot filler notation to identify actors, actions and objects. Additional work could also be employed to fill a case slot which would provide greater context to the system and therefore more accurate semantic definitions.

TODO mention semantic meaning aggragation for gramma constructs like NP

8 Conclusion

References

Lynch, S.: 2016, Lkit, *LKit: A Toolkit For Natural Language Interface Construction* .

Manning, ChristopherSchutze, H.: 1999, Foundations of statistical natural language.

NLTK: 2016, `http://www.nltk.org/`.