Automatic Generation of Wide-Coverage Semantic Representations in NLTK

B057443

Master of Science

Speech and Language Processing

School of Philosophy, Psychology and Language Sciences

University of Edinburgh

2015

Abstract

Existing semantic parsing systems are largely data-driven models that suffer a limitation in coverage from a lack of annotated training data. Increasing the coverage of semantic parsers is important for scaling systems to perform open-domain tasks related to natural language understanding (NLU) such as question-answering and information-retrieval. This thesis presents a system for determining ungrounded, wide-coverage semantic representations from CCG parses without additional training data. The system is modelled on the GRAPHPARSER system implemented by Reddy et. al. 2014. This is preliminary work towards an open-domain semantic parser in NLTK. It is implemented in the Python programming language within the NLTK framework.

Acknowledgements

I would like to thank my supervisor Ewan Klein for his advice on the design of the system and for imparting his knowledge of computational semantics and working with NLTK. I would also like to thank Siva Reddy for teaching me the inner workings of GRAPHPARSER and for his advice on the implementation.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*B057443*)

Table of Contents

Chapter 1

Introduction

1.1 Semantic Parsing and the Scalability Problem

Tasks such as question answering and information retrieval require machines to understand the semantics of natural language input. Understanding can be broken into two subfields: shallow semantic analysis (e.g. semantic role labelling, word-sense disambiguation) and deep semantic analysis. Semantic parsing is a form of deep semantic analysis which maps natural language input onto a meaning representation (MR), which is a formal, machine-readable representation of the semantics of the input. Early work on semantic parsing focused on providing natural language interfaces to specific databases [Androutsopoulos et al., 1995, Miller et al., 1996, Zelle and Mooney, 1996]. These approaches were largely rule-based and engineered for specific domains with well-defined concepts. Furthermore, they mapped NL input directly onto executable queries. The SCISSOR, WASP, and KRISP systems adopted statistical approaches to learning the translation from NL to directly executable MR, and thus could scale better. Still, they relied upon hand-annotated datasets of NL-MR pairs, which are costly to create [Ge and Mooney, 2005, Wong and Mooney, 2006, Kate and Mooney, 2006].

Later work attempted to reduce the dependence on annotated training data through unsupervised approaches. The approaches of [Poon and Domingos, 2009] and [Titov and Klementiev, 2011] treat the problem of mapping words and phrases to MR predicates as a clustering task. Here, examples from the NL are clustered together such that each cluster corresponds to a predicate or set of predicates in the MR. This approach does not require any annotated training data. Other approaches used a feedback signal in order to learn the correct mapping from NL to MR [Clarke et al., 2010, Goldwasser et al., 2011]. This approach does not eliminate the need for annotated

training data, but the data required is much easier to obtain: NL questions paired with gold-standard answers from a database.

Training data is not, however, the only impediment to the scalability of semantic parsers: the output MR of a semantic parser must be *grounded*. That is, the MR must have some well-defined meaning or function, e.g. a field in a database. Approaches such as [Kwiatkowski et al., 2013] and [Reddy et al., 2014] first parse to an ungrounded logical form using combinatory categorial grammar (CCG), and a mapping is learnt from these ungrounded forms to the target meaning representation language (MRL). This approach allows the same grammar structure to be used across domains. Thus the only theoretical limitation to the grounding of the semantic parser's output is the available ontologies and databases. Additionally, the use of CCG for semantic parsing is advantageous due to CCG's transparent interface between syntax and semantics. This is discussed in more detail in the next section.

1.2 Combinatory Categorial Grammar

Combinatory Categorical Grammar (CCG) is a syntactic grammar formalism developed as an extension of categorial grammars [Steedman, 2000]. In categorial grammars, the syntactic type of constituents is defined as either an argument or a function that takes arguments [Steedman and Baldridge, 2011]. Those that are arguments are called *primitive categories* and those that are functions are called *complex categories*. In contrast to context-free grammars that require explicit production rules (e.g. $S \rightarrow NP\ \ VP$), possible combinations of categories in categorial grammar are encoded in the categories themselves. Complex categories specify the type and direction of their arguments and the type of their result using the notation $\alpha \backslash \beta$, where α is the range of the function and β is the domain [Steedman, 1996]. The slash notation (e.g. $\backslash$or $/$) indicates the direction of the argument. $\backslash$ indicates that the argument is to the left and $/$ indicates that it is to the right. For example, the complex category

$$(S \backslash NP)/NP$$

first takes an argument of type NP to its right and then another of type NP to its left. It returns the primitive category S. The parentheses specify the order in which the function can be applied to its arguments.

Categorial grammar defines two functions for combining categories:

Forward application ($>$)

$X/Y \quad Y \quad \Rightarrow \quad X$

Backward application ($<$)

$Y \quad X \backslash Y \quad \Rightarrow \quad X$

The use of only these two rules limits the expressive power of categorial grammar to match that of context-free grammar [Steedman and Baldridge, 2011]. CCG increases the expressive power over categorial grammar by defining additional combinatory rules. These are:

Forward composition ($B_>$)

$X/Y \quad Y/Z \quad \Rightarrow \quad X/Z$

Backward composition ($B_<$)

$Y \backslash Z \quad X \backslash Y \quad \Rightarrow \quad X \backslash Z$

Forward substitution ($S_>$)

$(X/Y)/Z \quad Y/Z \quad \Rightarrow \quad X/Z$

Backward substitution ($S_<$)

$Y \backslash Z \quad (X \backslash Y) \backslash Z \quad \Rightarrow \quad X \backslash Z$

Forward type-raising ($T_>$)

$X \quad \Rightarrow \quad T/(T \backslash X)$

where $X \in \{NP\}$

Backward type-raising ($T_>$)

$X \quad \Rightarrow \quad T \backslash (T/X)$

where $X \in \{NP, PP, S\}$

CCG is a lexicalized grammar formalism. This means that each lexical item (word) is assigned a category, and the syntactic analysis of a sentence is a combination of these categories. CCG is thus defined by a lexicon and a rule set. The lexicon consists of words paired with categories. The rule set is the set of combinatory rules such as those described above. Thus parsing with CCG can be broken into two steps: 1) assign each token in the input a category from the lexicon; 2) combine these categories via the rules in the rule set. Combination proceeds until the start symbol is reached and the

parse covers the entire input sentence. An example CCG parse is shown in figure 1.1.

$$\dfrac{\dfrac{\overline{\textit{John}}}{NP} \quad \dfrac{\dfrac{\overline{\textit{sees}}}{(S\backslash NP)/NP} \quad \dfrac{\overline{\textit{Mary}}}{NP}}{S\backslash NP}\,{>}}{S}\,{<}$$

Figure 1.1: CCG parse for "John sees Mary".

CCG also extends the categorial grammar formalism by treating categories as encoding a semantic representation. The parse in figure 1.1 can thus be annotated with the semantic representations of its constituents, via the colon operator. This is shown in figure 1.2.

$$\dfrac{\dfrac{\overline{\textit{John}}}{NP : john'} \quad \dfrac{\dfrac{\overline{\textit{sees}}}{(S\backslash NP)/NP : see'} \quad \dfrac{\overline{\textit{Mary}}}{NP : mary'}}{S\backslash NP : see'mary}\,{>}}{S : see'mary'john'}\,{<}$$

Figure 1.2: CCG parse for "John sees Mary" annotated with semantic representations.

The semantic representations combine according to the same rules as the categories. This transparency between syntax and semantics is a major advantage of using CCG for semantic parsing over other grammar formalisms.

1.3 Goals of the Project

This thesis presents a system for automatically parsing natural language input into logical form without annotated training data. The system aims for wide-coverage yet ungrounded representations. By wide-coverage it is meant that natural language input in many domains can be assigned a semantic representation. By ungrounded it is meant that the atoms that comprise the representation do have any meaning *per se*. That is, the representation cannot be used directly, e.g. to query a database.

The system is implemented as a module in the Natural Language Toolkit (NLTK) [Bird et al., 2009]. NLTK is an extensible framework for natural language processing in Python. It supports many NLP tasks such as tokenization, tagging, syntactic parsing, and classification. Currently, support for mapping from syntax to semantics is limited

to a few hand-built feature-based grammars (cf. `sem{0,1,2}.fcfg`, `event.fcfg`) which can be parsed with the `nltk.parse` module. The goal of this project is to extend this functionality by implementing a system which automatically determines semantic representations from syntax. There are two subgoals of this implementation:

1. The system should be practical: Users should be able to use the output for tasks that require deep semantic analysis.

2. The system should be able to be used as a teaching tool for semantic parsing: It should provide a transparent mapping between syntax and semantics. The steps in a given semantic derivation should be representable. The formalism used for representing the semantics should be relatively human-readable.

The implementation of the system is based on the GRAPHPARSER semantic parser developed by [Reddy et al., 2014]. GRAPHPARSER is a large-scale semantic parser that works by parsing natural language input into an ungrounded semantic representation and then learning a mapping from these representations to a target query language. The system implemented here accomplishes the first part of this task within NLTK, i.e. mapping natural language input to ungrounded semantic representations.

The structure of this thesis is as follows. Chapter 2 details the design of the system. First the requirements that the system must fulfill are defined. The design of the two main components are then discussed in detail. Of particular interest is section 2.3.2, which presents the procedure for automatically determining semantic representations for tokens from their assigned CCG categories. This chapter also discusses how these are composed into a semantic representation for a full input sentence.

Chapter 3 provides the implementation details. The chapter first outlines the tools used that already exist in NLTK. Then, the implementation of each component described in chapter 2 is discussed.

Chapter 4 describes the evaluation of the system. This section gives the coverage of the parser over the GEOQUERY880 dataset as well as an evaluation of the correctness of the parser's output.

Chapter 5 summarises and outlines future work.

Chapter 2

Design

The semantic parser is broken into two main components: a syntactic component and a semantic component. Input to the syntactic component is a natural language sentence. This sentence is parsed using a CCG parser into a syntactic parse tree. This tree is then passed to the semantic component, which first determines the semantics for the tokens in the sentence and then composes them following the syntactic parse. The lexical semantics are thus determined on the fly. This differs from most other approaches to semantic parsing in which a lexicon mapping words to semantic representations is learned first, and then used in parsing. The output of the system is a formal expression representing the semantics of the input sentence. Figure 2.1 summarizes the structure of the overall system.

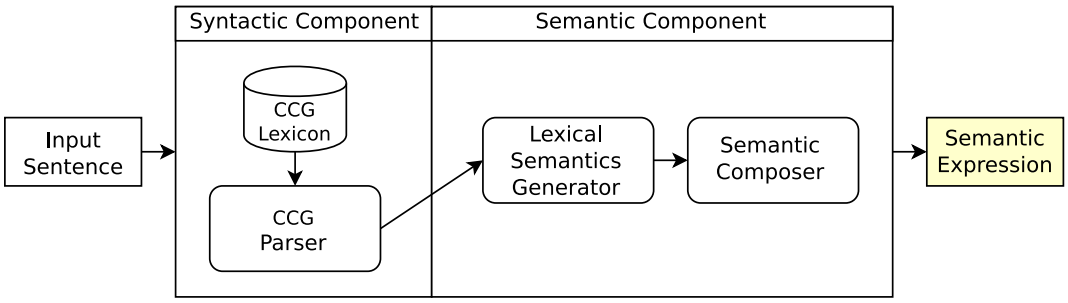

Figure 2.1: Overall structure of the semantic parser.

This chapter discusses the overall design of the semantic parser. First, I enumerate the requirements that the semantic parser must fulfill. Then I discuss the syntactic and semantic components in turn, explaining how each requirement is addressed, as applicable. Chapter 3 discusses how this design is implemented and used in the NLTK framework.

2.1 Requirements

The system has the following functional requirements. It should

- be able to build a semantic representation for practically any input sentence (wide coverage).
- compose a semantic representation of an input sentence from the semantic representations of its constituent words (principle of compositionality).

In order to meet these requirements the system must be able to

1. Assign basic semantic representations to the input tokens.
2. Compose these basic semantic representations into complex semantic expressions.

It should be noted that the requirements above are requirements to be met by the *semantic* component, not the syntactic. Section 2.2 will describe how the syntactic component facilitates the fulfillment of these requirements by the semantic component.

2.2 Syntactic Component

The syntactic component is composed of a CCG syntactic parser and a CCG lexicon. The input to the syntactic component is a natural language sentence. The output is a syntactic parse tree, which is passed to the semantic component. The CCG category for each token in the input is looked up in the CCG lexicon. These categories are then combined using the CCG combinators described in section 1.2 until *S* is obtained at the root node of the parse tree. The purpose of the syntactic component is to provide a guide for composing the semantics. While a variety of syntactic formalisms can be used for this purpose, CCG was chosen for two reasons:

1. CCG syntactic categories correspond to λ expressions. This facilitates the lexical semantics generation step. This is because complex CCG categories define functions which take arguments. The process by which complex categories are translated into λ forms is described in section 2.3.2.

2. The combinatory rules described in section 1.2 correspond to functional operations. Thus CCG categories and λ forms combine according to functions of the same type. This is illustrated below.

   Application
   $X/Y \;>\; Y \;\Rightarrow\; X$
   $\lambda y.F(y)(x) \;\Rightarrow\; F(x)$
   The type of the application function for both the CCG category and the λ function is $Y \to X$.

   Composition
   $X/Y \;B_{>}\; Y/Z \;\Rightarrow\; X/Z$
   $\lambda y.F(y) \;\circ\; \lambda z.G(z) \;\Rightarrow\; \lambda x.F(G(x))$
   The type of both functions is $(Y \to X) \to (Z \to Y) \to Z \to X$.

   Substitution
   $(X/Y)/Z \;S_{>}\; Y/Z \;\Rightarrow\; X/Z$
   $\lambda zy.F(z,y) \;\$\; \lambda z.G(z) \;\Rightarrow\; \lambda z.F(z,G(z))$
   Where \$ denotes the substitution operation.
   The type of both functions is $(Z \to Y \to X) \to (Z \to Y) \to Z \to X$.

   Type-raising
   $T_{>} \;X \;\Rightarrow\; Y/(Y \setminus X)$
   $x \;\Rightarrow\; \lambda f.f(x)$
   The type of both function is $X \to (X \to Y) \to Y$.

Thus the syntactic component will help the semantic component fulfill requirement 1 (assigning lexical semantics) because each CCG category has a corresponding λ form. It will help fulfill requirement 2 (compositionality) because the CCG parse tree specifies the order and type of combinatory operations, and these reflect functional operations.

2.3 Semantic Component

The input to the semantic component is the output of the syntactic component, i.e. a syntactic parse tree. The semantic component uses the parse tree to determine a semantics for the input sentence. It is a two-part system: a lexical semantics generator and a semantic composer. The lexical semantics generator determines the semantic

representations for tokens in the input sentence. The semantic composer combines these semantic representations into an expression that represents the semantics of the full input sentence.

This section is broken into three parts: First I describe the semantic formalism used; then I describe the lexical semantics generation procedure; finally I describe the semantic composition algorithm.

2.3.1 Semantic Formalism

The semantic formalism employed here is, following [Reddy et al., 2014], Neo-Davidsonian event semantics with lambda abstraction. Davidsonian semantics is a form of first-order logic that uses event identifiers (e) in order to refer to verb predicates and their arguments. Neo-Davidsonian semantics extends this by breaking arguments into thematic roles, e.g. *agent* and *patient*, linked by conjunction [Parsons, 1990]. This extension allows the formalism to deal with grammatical phenomena such a passive case, in which grammatical roles do not reflect semantic roles. An example Neo-Davidsonian representation is given below.

Brutus stabbed Caesar with a knife.
$\exists e.stab(e) \wedge agent(e, Brutus) \wedge patient(e, Caesar) \wedge with(e, knife)$

This semantic formalism was chosen for the following reasons:

- The reification of events by means of event identifiers (e) allows the formalism to more gracefully express two linguistic phenomena: the adverbial modification of events, e.g. $stab(e) \wedge quickly(e) \wedge quietly(e)$ vs. $quietly(quickly(stab()))$; variable number of arguments to verbs, e.g. $with(e, knife)$ in the logical expression above is an optional argument, as $stab(e) \wedge agent(e, Brutus) \wedge patient(e, Caesar)$ is still a meaningful expression.

- Neo-Davidsonian semantics is a minimal recursion semantics or *flat* semantics. This means that predicates can not be embedded within each other, but are rather join by logical conjunction. This simplifies the semantic composition procedure because there is no need to backtrack to modify an expression. Rather, an expression is modified by simply conjoining a predicate modifying the variable in question. For example, consider the sentence "Brutus stabbed Casesar quickly". A non-flat representation would express the adverbial modification of "stabbed"

by modifying the expression for "stabbed" directly:

$$quickly(stabbed(Brutus, Caesar))$$

This would require, during the composition process, the parser to backtrack to find the *stabbed* predicate once it encountered "quickly" in order to modify it. A flat semantics, on the other hand, does not need to backtrack; it simply joins by logical conjunction an expression modifying the event variable:

$$\exists e.stab(e) \wedge agent(e, Brutus) \wedge patient(e, Caesar) \wedge quickly(e)$$

This requires no backtracking. $quickly(e)$ can simply be appended to the expression.

- Neo-Davidsonian semantics is a type of first-order logic that can be extended by lambda-abstraction. A first-order logic parser, including functions to handle the λ-calculus and event variables, is already implemented in NLTK in the `logic.sem` package.

2.3.2 Automatic Generation of Lexical Semantics

Section 1.2 described how complex syntactic categories in CCG represent combinatorial functions, which define the number, type, and direction of the arguments and the type of the result. The purpose of this section is to describe the process by which these combinatorial functions are translated into functions in the λ-calculus, fulfilling requirement 1. This section is an expansion of the process discussed in the appendix of [Reddy et al., 2014]. The result of the semantic generation step is a *semantic category*, which can be understood in terms of two components: 1) a functional definition, which corresponds to the number and type of arguments specified in the syntactic category; 2) a semantic definition, which is expresses the meaning of the token for which a semantic category is being constructed. The functional definition is what ensures that the semantic categories are correctly composed, and the semantic definition makes the composed semantic expression interpretable.

Construction of the semantic category proceeds in two steps: First, a stem expression is constructed from the syntactic category. This is the functional definition. Next,

token-specific subexpressions are added based on the semantic category of the input token. This is the semantic definition. This process is summarized in figure 2.2. This section first describes the process of constructing the stem expression. Then, it outlines the rules required for determining a token's semantic type and semantic definition.

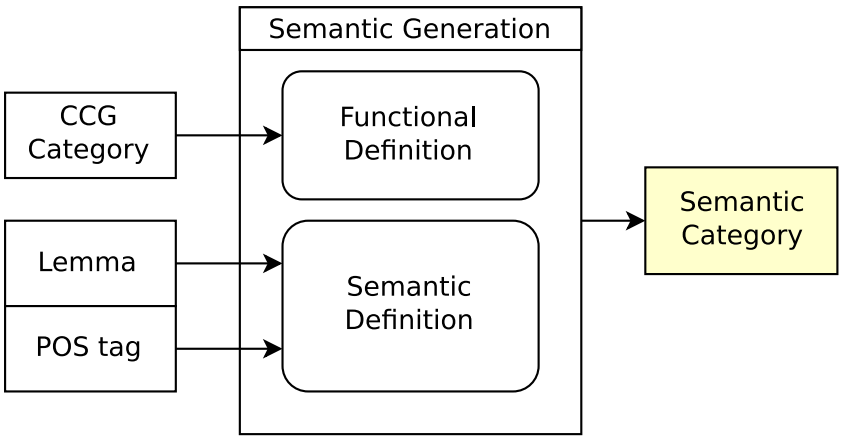

Figure 2.2: The process by which a semantic category is generated.

2.3.2.1 Functional Definition

The stem expression, or functional definition, defines the number and type of the arguments of the semantic category. The construction of the stem expression is guided by the information provided by the syntactic category assigned to the input token by the CCG lexicon. This information is, as mentioned above, the number and type of the arguments to the syntactic category.

The number of arguments of the semantic category is equal to the number of arguments of syntactic category. To use an example, "admires" has the syntactic category $(S \setminus NP)/NP$, which takes two NP arguments. Thus the number of arguments of the semantic category is 2. The definition of the arguments of the semantic category is related to the syntactic type of the arguments to the syntactic category. In the "admires" example, the syntactic type of the first argument is NP. NP (noun phrase) corresponds to an entity. In the Neo-Davidsonian semantics used here, entities are represented as functions that take a single variable as an argument, e.g. an equality function such as "John" :: $\lambda x.x = John$. The first argument to the semantic category is, then, a function that takes a single argument. The second argument, being of syntactic type NP, is also a function that takes a single argument. Thus the semantic category takes two functions as arguments. Each of these argument functions operates on a variable, e.g. x in $\lambda x.x = John$.

The basic syntactic category, i.e. $(S \setminus NP)/NP$, does not specify whether the two NP's have the same referent. If both NP's have the same referent, then they will operate on the same variable in the final semantic category. If they do not have the same referent, they will operate on different variables. Consider "admires" in the following parse:

$$\dfrac{\dfrac{\underline{John}}{NP} \quad \dfrac{\dfrac{\underline{admires}}{(S\backslash NP)/NP} \quad \dfrac{\underline{Mary}}{NP}}{S\backslash NP}>}{S}<$$

Here the first NP argument has "Mary" as its referent and the second has "John". Because they have different referents, they must operate on different variables in the final semantic category. By contrast consider "also" in the following parse:

$$\dfrac{\dfrac{\underline{John}}{NP} \quad \dfrac{\dfrac{\underline{also}}{(S\backslash NP)/(S\backslash NP)} \quad \dfrac{\dfrac{\underline{admires}}{(S\backslash NP)/NP} \quad \dfrac{\underline{Mary}}{NP}}{S\backslash NP}>}{S\backslash NP}>}{S}<$$

Here, although the syntactic category for "also" has two NP's, both have the same referent, namely "John". In this case, the NP arguments will operate on the same variable.

Referent information for a given syntactic category is obtained by translating the syntactic category into its corresponding indexed syntactic category using a predefined mapping. The indexed syntactic category for "admires" :: (S\NP)/NP) is $(S_e \setminus NP_x)/NP_y$. The subscripts are variable names for the referents of the primitive category to which they are attached. Thus for "admires", each NP has a different variable name and thus a different referent. For "also", however, the indexed syntactic category is $(S_e \setminus NP_x)/(S_e \setminus NP_x)$ and so both NP's have the same referent and will operate on the same variable in the final semantic category.

Referent information, alongside number and type of arguments, is all that is required to construct the functional definition of the semantic category. The following procedure is employed for translating an indexed syntactic category into a functional definition. The "admires" example will be used throughout to illustrate the process.

1. The indexed syntactic category is parsed according to the parentheses and back and forward slashes into a binary tree representing the curried function. In the binary tree each non-leaf node is a function. Its right-hand child is its argument and its left-hand child is its return value. For example, the indexed syntactic category for "admires" is $(S_e \setminus NP_x)/NP_y$. The tree parse for this syntactic category is shown in figure 2.3. Since the tree represents the curried function, the left-hand child (return value) can be another function. Thus the root node in figure 2.3 takes N_y as its argument and returns the function $S_e \setminus NP_x$.

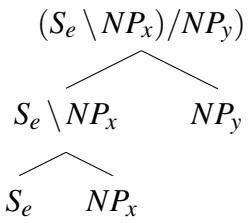

Figure 2.3: Parse of the indexed syntactic category $(S_e \setminus NP_x)/NP_y$.

2. The functions and arguments in the parse tree are assigned variable names. These variable names are those that will be used in the resulting λ expression. Function variables are uppercase letters and argument variables are lowercase. Figure 2.4 shows an example assignment. The type of this function is $y \to x \to e$.

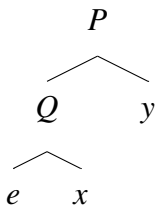

Figure 2.4: Parse tree from figure 2.3 assigned variable names.

3. The parse tree is traversed top-down and the function variables are paired with their arguments. P is paired with its argument y and Q with x. The result of this step is an ordered set of predicate variables paired with their argument variables as well as an individual variable. The individual variable is the left-most leaf in the parse tree. It is the argument to the function that is finally returned, in this case e.

4. This ordered pairing is then used to construct a lambda expression. First the predicate lambda terms are added, then the individual variable. All arguments

are then existentially quantified. Finally, the body of the expression is constructed by conjoining the subexpressions formed by each predicate variable and its arguments. Thus the functional definition of the semantic category for the example above is $\lambda P \lambda Q \lambda e.\exists y \exists x.[P(y) \wedge Q(x)]$.

In this way the functional definition is built. Appendix .1 discusses this process for a more complex CCG category.

2.3.2.2 Semantic Definition

While the functional definition of the semantic category governs how it is able to combine with other semantic categories, the semantic definition is what makes the semantic category interpretable. In other words, the semantic definition describes what the semantic category actually *means*. The semantic definition is determined according to a token's lemma and POS tag. This information is sufficient to assign a *semantic type*. The semantic type is then used to construct the semantic definition. Table 2.1 gives the rules for determining a semantic type from a lemma and POS tag and for constructing a semantic definition.

Lemma	POS Tag	Semantic Type	Example Semantic Category
be	*	COPULA	$\lambda P \lambda Q \lambda y.\exists z.[(z = y) \wedge P(z) \wedge Q(y)]$
the	*	UNIQUE	$\lambda P \lambda y.[P(y) \wedge \text{UNIQUE}(y)]$
not	*	NEGATION	$\lambda P \lambda Q \lambda e.\exists z.[P(e,y) \wedge Q(z) \wedge \text{NEGATE}(y)]$
no	*	NOTEXISTS	$\lambda P \lambda y.[P(y) \wedge NO(y)]$
*	NN, NNS	TYPE	$\lambda x.[tok(x)]$
*	NNP*, PRP*	ENTITY	$\lambda x.[x = tok]$
*	VB*, IN, TO, POS	EVENT	$\lambda P \lambda Q \lambda e.\exists x \exists y.[P(y) \wedge Q(x) \wedge$ $tok.agent(e,x) \wedge tok.patient(e,y)]$
*	JJ*	TYPEMOD	$\lambda P \lambda y.[P(y) \wedge tok(y)]$
*	RB*	EVENTMOD	$\lambda P \lambda e.\exists x.[P(e,x) \wedge tok(e)]$
*	CD	COUNT	$\lambda P \lambda y.[P(y) \wedge COUNT(y,tok)]$
*	WP*, WDT, WRB	QUESTION	$\lambda P \lambda e.\exists x.[P(e,x) \wedge TARGET(x)]$

Table 2.1: Rules for constructing semantic categories from lemma and POS tag. Adapted from [Reddy et al., 2014]. Where *tok* appears in a semantic category, it stands for the token that the semantic category represents. E.g. "John" (NNP) :: ENTITY :: $\lambda x.[x = \texttt{John}]$.

The semantic definition is the subexpression of the semantic category that occurs after the stem expression (functional category). E.g. for the semantic type EVENT, the

functional definition is $\lambda P \lambda Q \lambda e. \exists x \exists y. [P(y) \wedge Q(x)]$ and the semantic definition is $tok.agent(e,x) \wedge tok.patient(e,y)$ (where tok is a placeholder for the token to which this semantic category is assigned). The full expression is constructed by joining by logical conjunction the stem expression and the semantic definition to form $\lambda P \lambda Q \lambda e. \exists x \exists y. [P(y) \wedge Q(x) \wedge tok.agent(e,x) \wedge tok.patient(e,y)]$.

The exact form of the semantic definition depends upon the semantic type as well as the stem expression. The semantic type determines which predicates will be used, e.g. the $NEGATE$ predicate for semantic type NEGATION. The arguments to these predicates are determined by the arguments in the stem expression. To use an example, the verb "admires" has the indexed syntactic category $(S_e \setminus NP_x)/NP_y$. The functional definition of this syntactic category is $\lambda P \lambda Q \lambda e. \exists y \exists x. [P(y) \wedge Q(x)]$. This semantic category thus takes two arguments, y and x, and returns a function of the event variable e. The semantic type is EVENT, thus, according to the rules in table 2.1 the semantic definition will consist of a conjunction of the verb predicate with each of its arguments. The first argument is y. Note that in the syntactic category the first argument is on the right-hand side (indicated by the forward slash). In English, for verbs with two arguments, the right-hand side argument is the direct object, e.g. "Mary" in "John admires Mary." and the second argument, on the left-hand side, is the subject, e.g. "John". Thus y is the direct object and x is the subject, so the semantic definition is $admires.agent(e,x) \wedge admires.patient(e,y)$.

In this way the semantic category is constructed. To recapitulate, input to the semantic component is a syntactic parse tree, including the token (which is analysed into its lemma), the POS tag, and syntactic category. The syntactic category is translated into an indexed syntactic category and then parsed into a binary tree representing it as a curried function. Argument variables are assigned to leaf nodes and predicate variables are assigned to non-leaf nodes. The tree is then traversed top-down and predicate variables are paired with their arguments. These pairings are used to construct the functional definition of the semantic category. Next, a semantic type is assigned, according to the rules in table 2.1. The semantic type, along with the number and order of arguments in the functional definition, are used to construct the semantic definition. The functional definition and the semantic definition are then joined by logical conjunction to form the final semantic category.

2.3.3 Semantic Composition

This section describes how semantic categories determined in the lexical semantics generation step are composed to form the full semantic representation of the input sentence, fulfilling requirement 2 (compositionality). The input to the semantic composer is a CCG syntactic parse tree and the output is a semantic representation of the sentence. The semantic composer has two elements: 1. a recursive, depth-first tree traversal algorithm; 2. a set of rules for combining subexpressions. The tree traversal algorithm operates over the CCG parse tree, and is what guides the composition. The combinatory rule set is what governs how two subexpressions combine.

Algorithm 1 Semantic composition algorithm

```
1: procedure COMPOSE(root)
2:     if root = leaf then
3:         return semanticCategory(root)
4:     else                                   ▷ Recursively build subexpressions
5:         left_expression ← COMPOSE(leftChild(root))
6:         right_expression ← COMPOSE(rightChild(root))
7:         return applyRule(left_expression, right_expression)
8:     end if
9: end procedure
```

Algorithm 1 outlines the composition algorithm. The algorithm takes the root node of a binary parse tree as input and recursively combines the subexpressions corresponding to the left and right children to obtain a full semantic representation at the root *S* node. Here we see how the semantic component determines the semantic representations for the tokens on the fly. When the algorithm reaches a leaf node, i.e. a token, the function *semanticCategory* (line 3) constructs a semantic category for that token according to the process described in section 2.3.2. When the algorithm reaches a non-leaf node, it combines the subexpressions for its left and right children according to the CCG combinatory rule set. Each non-leaf node in the CCG parse tree specifies one of the CCG combinatory rules described in section 1.2. These combinatory rules correspond to functional operations, according to which the subexpressions are combined. This operation (line 7) is a table lookup of the functional operation to which the CCG combinator is mapped. This mapping is shown in table 2.2.

Figure 2.5 gives an example semantic derivation for the sentence “John admires

CCG combinator	Functional operation	Functional Definition
Forward Application (>)	Function application	$l(r)$
Backward Application (<)	Function application	$r(l)$
Forward Composition ($B_>$)	Function composition	$\lambda x.l(r(x))$
Backward Composition ($B_<$)	Function composition	$\lambda x.r(l(x))$
Forward Substitution ($S_>$)	Function substitution	$\lambda x.l(x,r)$
Backward Substitution ($S_<$)	Function substitution	$\lambda x.r(x,l)$
Type-Raising (T)	Function type-raising	$\lambda f.f(l)$

Table 2.2: The mapping from CCG combinator to functional operation. In the functional definition column, l is the left-hand subexpression, r is the right-hand subexpression, f is a predicate variable, and x is an individual variable.

Mary". The leaf nodes are first assigned their semantic category by the lexical semantics generation system. These are then combined according to the CCG parse. First the semantic expression for "admires" is applied to that for "Mary", forming the subexpression $\lambda Q\lambda e.\exists x\exists y.[(x = Mary) \wedge Q(y) \wedge admires.agent(e,y) \wedge admires.patient(e,x)]$. This subexpression is then applied to the semantic expression for "John", yielding $\lambda e.\exists x\exists y.[(x = Mary) \wedge (y = John) \wedge admires.agent(e,y) \wedge admires.patient(e,x)]$at the root.

$$\dfrac{\dfrac{\textit{John}}{\begin{array}{c}\text{NP}\\ \lambda x.[x = John]\end{array}} \quad \dfrac{\textit{admires}}{\begin{array}{c}\text{(S\NP)/NP}\\ \lambda P\lambda Q\lambda e.\exists x\exists y.[P(x) \wedge Q(y) \wedge\\ admires.agent(e,y) \wedge admires.patient(e,x)]\end{array}} \quad \dfrac{\textit{Mary}}{\begin{array}{c}\text{NP}\\ \lambda x.[x = Mary]\end{array}}}{\begin{array}{c}\text{S\NP}\\ \lambda Q\lambda e.\exists x\exists y.[(x = Mary) \wedge Q(y) \wedge\\ admires.agent(e,y) \wedge admires.patient(e,x)]\end{array}}>}{\begin{array}{c}\text{S}\\ \lambda e.\exists x\exists y.[(x = Mary) \wedge (y = John) \wedge\\ admires.agent(e,y) \wedge admires.patient(e,x)]\end{array}}<$$

Figure 2.5: Example semantic derivation, overlaid on the CCG parse.

2.4 Summary

This chapter described the design of the semantic parsing system. It is made up of two main components: a syntactic component and a semantic component. The syntactic

component is described in section 2.2. It consists of a CCG syntactic parser and and CCG lexicon. The input to the syntactic component is a natural language sentence. The CCG parser parses the sentence into a parse tree which is then passed to the semantic component. The semantic component is further broken down into two parts, an lexical semantics generator and a semantic composer. Practically, these work in tandem: the semantic composer is a depth-first traversal of the syntactic parse tree. When it reaches a leaf node, the lexical semantics generator determines the semantic category for the token at that leaf node based on the token's lemma, POS tag, and indexed syntactic category. These semantic categories are then composed according to the parse tree and the functional operations specified on each non-leaf node, until the algorithm reaches the root node of the tree, the composed expression at which is the semantic representation of the input sentence.

Chapter 3

Implementation

This chapter describes the implementation in NLTK of the system design laid out in chapter 2. Section 3.1 discusses what existing NLTK packages were used. Section 3.2 details the functional architecture of the system and the class definitions. Section 3.3 describes the usage of the semantic parser.

3.1 Existing Tools

Two existing NLTK packages were used in the implementation of the semantic parser: the λ-calculus processor and the CCG parser.

3.1.1 The λ-calculus Processor

The λ-calculus processor is implemented in the `nltk.sem.logic` module. It is used for building and composing logical expressions. It consists of the `Expression` class, its subclasses, and associated methods for manipulating logical expressions. There are a number of subclasses of `Expression`, but the ones used here are `LambdaExpression`, `ExistsExpression`, `AndExpression`, and `ApplicationExpression`.

`LambdaExpression` is used for logical expressions with λ terms, so this class represents the Neo-Davidsonian logical forms associated with semantic categories for a token, e.g. "John" :: $\lambda x.[x = John]$ is a `LambdaExpression`.

`ExistsExpression` represents logical expressions that have no λ terms but have existentially quantified variables, e.g. $\exists x.[F(x)]$. `AndExpression` represents logical expressions with no λ terms and no quantified variables but with logical conjunction, e.g. $[F(x) \wedge G(x)]$. `ExistsExpression` and `AndExpression` are only used at an inter-

mediate step in building the logical form for a token.

Finally, `ApplicationExpression` represents a logical expression which has been applied to an argument, e.g. $\lambda P.\exists x.[P(x)](\lambda x.[x = John])$ and its β-reduced form $\exists x.[(x = John)]$. This class is used in the composition algorithm when combining expressions.

`Expression` objects are organized recursively. The tree in figure 3.1 represents the organization of the `LambdaExpression` $\lambda P \lambda Q \lambda x.\exists y.[P(x) \wedge Q(y)]$.

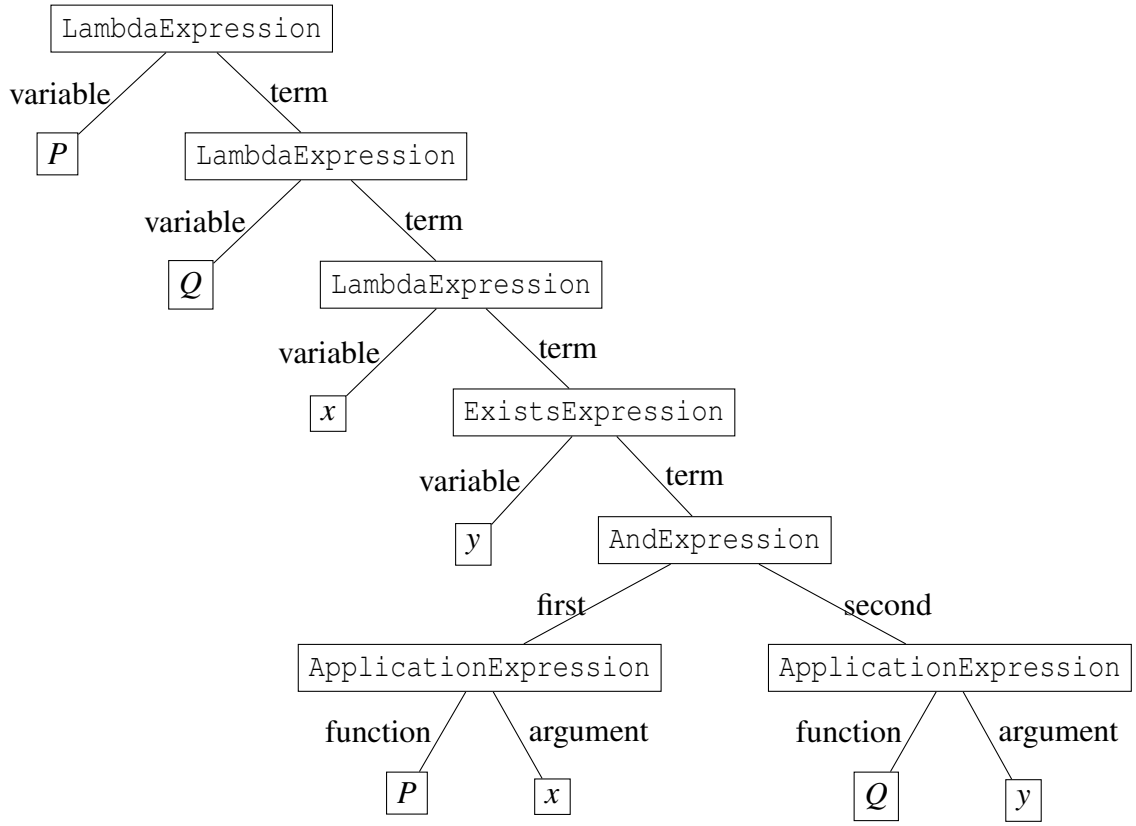

Figure 3.1: Tree representing the recursive organization of `LambdaExpression`.

The `sem.logic` module allows expressions to be constructed from their ascii representation. The following illustrates the ascii representation of terms:

```
λ  =  \
∃  =  exists
∧  =  &
```

Thus the ascii representation of $\lambda P \lambda Q \lambda x.\exists y.[P(x) \wedge Q(y)]$ is
`\P \Q \x. exists y.(P(x) & Q(y))`. The λs can be shorthanded to
`\P Q x. exists y.(P(x) & Q(y))`.

To create an `Expression` object from the ascii representation, the `Expression.fromstring()` method is used, e.g.

```
expr = Expression.fromstring(r'\P Q x. exists y.(P(x) & Q(y))')
```

The method automatically determines the appropriate subclass of `Expression`. Thus in this case `expr` will be an instance of `LambdaExpression`.

To perform functional application, the `ApplicationExpression` class is used. The constructor for this class takes two arguments that are of type `Expression`. The constructor will apply the first argument to the second. E.g.

```
>>> expr1 = Expression.fromstring(r'\P y.P(y) & woman(y)')
>>> expr2 = Expression.fromstring(r'\x.(woman(x))')
>>> appex = ApplicationExpression(expr1, expr2)
>>> print appex
(\P y.(P(y) & human(y))(\x.woman(x))
```

β-reduction is performed with the `ApplicationExpression.simplify()` method.

```
>>> print appex.simplify()
\y.(woman(y) & human(y))
```

While the λ calculus processor supports functional application, it does not support other functional operations such as composition, substitution, and type-raising.

3.1.2 CCG Parser

The CCG parser is implemented in the `nltk.ccg` package. It is comprised of two main components: a parsing algorithm and a CCG lexicon. The parsing algorithm also requires a set of combinatory rules. The combinatory rules outlined in section 1.2 are supported. To reiterate, these are application, composition, substitution, and type-raising. These are defined in `nltk.ccg.combinator`.

The parsing algorithm used is the Cocke-Younger-Kasami (CYK) bottom-up chart

parsing algorithm. It is implemented in the `nltk.ccg.chart` module with the `CCGChartParser` class. The CYK algorithm works by examining all possible spans of words on the input and inserting edges where these spans can combine according to a combinatory rule. This process proceeds until the start symbol is reached. It is a chart parsing algorithm because possible combinations of sub-spans are stored in cells in a chart. An example chart is given in figure 3.2.

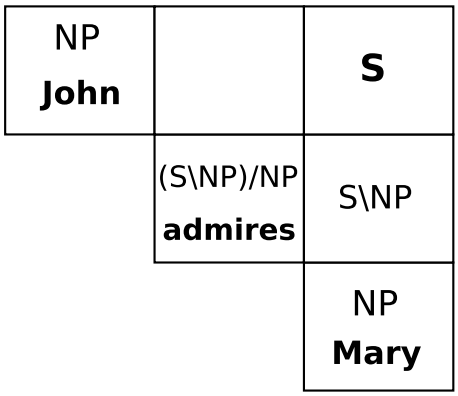

Figure 3.2: Example chart for the parse of "John admires Mary".

Thus the cell spanning "admires" and "Mary" holds a possible combination of their respective categories, i.e. $S \backslash NP$. Where no combination is possible, the cell is empty, e.g. the cell spanning "John" and "admires". The parse is complete when the cell spanning the start and end token of the input holds the start symbol, e.g. S. The use of backpointers allows all possible derivations to be stored in the same chart: a parse is retrieved by following the backpointers from the start symbol to the leaves. The output of `CCGChartParser.parse()` is an instance of `nltk.Tree`.

The CCG lexicon is implemented in `nltk.ccg.lexicon`. The class that represents the CCG lexicon is `CCGLexicon`. The class diagram for `CCGLexicon` is shown in figure 3.3.

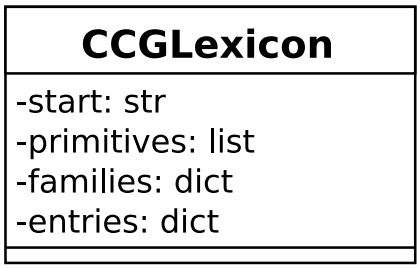

Figure 3.3: Class diagram for `nltk.ccg.CCGLexicon`.

There are four attributes of `CCGLexicon`. The `start` attribute is a string represent-

ing the start symbol of the grammar, usually `S`. `primitives` is a Python list holding all the primitive categories used by the lexicon, e.g. `[NP, VP, PP]`. `families` is a Python dictionary which maps a placeholder onto a primitive or complex category, e.g. `{TransVerb : (S\NP)/NP}`. `entries` is a Python dictionary holding the lexicon entries, e.g. `{John : NP}`.

There is a module-level function, `parseLexicon()`, which parses an ascii representation of a CCG lexicon into a `CCGLexicon` object. The ascii representation of a CCG lexicon must adhere to the following form:

- First the start symbol and primitive categories are defined with the format `:- <start symbol>, <primitive categories>`, where `<primitive categories>` is a comma-separated list of primitive categories. E.g. `:- S, NP, VP`.
- Second, any family definitions must be defined, one per line. These have the format `<replacement> :: <category>`. E.g. `TransVerb :: (S\NP)/NP`.
- Finally, categories are defined, one per line. These have the format `<word> => <category>`. E.g. `John => NP`.

An example of defining and instantiating a `CCGLexicon` is given below:

```
>>> ccglex = parseLexicon(r'''
...     :- S, NP
...     TransVerb :: (S\NP)/NP
...     I => NP
...     eat => TransVerb
...     peaches => NP
... ''')
```

3.2 Architecture

The semantic parser is implemented in `nltk.semparse`. The base class of the system is `SemanticParser`. The class diagram for `SemanticParser` is shown in figure 3.4.

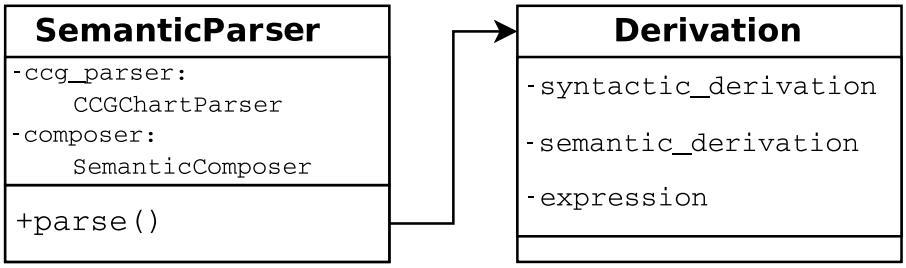

Figure 3.4: Class diagram for `SemanticParser`.

The `SemanticParser` class wraps instances of two classes: `CCGChartParser` and `SemanticComposer`. `SemanticParser` has only one method, `parse()`, which parses an input sentence into a semantic representation. Two sub-tasks comprise the `parse()` method: a call to `CCGChartParser` and a call to `SemanticComposer`. The `CCGChartParser.parse()` method first syntactically parses the input sentence. This parse is then passed to `SemanticComposer`, which composes the semantic representation of the input sentence according to the syntactic parse.

`SemanticParser.parse()` yields a `Derivation` object for each syntactic parse output by `CCGChartParser`. `Derivation` holds three types of data: a syntactic derivation, a semantic derivation, and a full logical expression. The syntactic derivation is the output of the `CCGChartParser`. It is an instance of `nltk.Tree` representing the syntactic parse of the input sentence. The semantic derivation is the output of the `SemanticComposer`. It is also an instance of `nltk.Tree` representing the steps in the derivation of the semantic parse. That is, the leaves of the tree are the semantic representations of the input tokens and the non-leaf nodes are the compositions of their children. An example semantic derivation for the sentence "John admires Mary" is shown in figure 3.5.

The root of the tree is the final composed semantic representation. The `expression` attribute of `Derivation` is thus equal to the root of the semantic derivation tree. Each node of the semantic derivation tree is an instance of the `nltk.sem.logic.Expression` class or some subclass thereof as described in section 3.1.1.

3.2.1 Composer Implementation

The `SemanticComposer` class is implemented in `nltk.semparse.composer`. This class is the implementation of algorithm 1 discussed in section 2.3.3. It has no externally visible attributes and only one externally visible method, `buildExpression()`. `buildExpression()` is the recursive tree traversal part of algorithm 1. `buildExpression()`

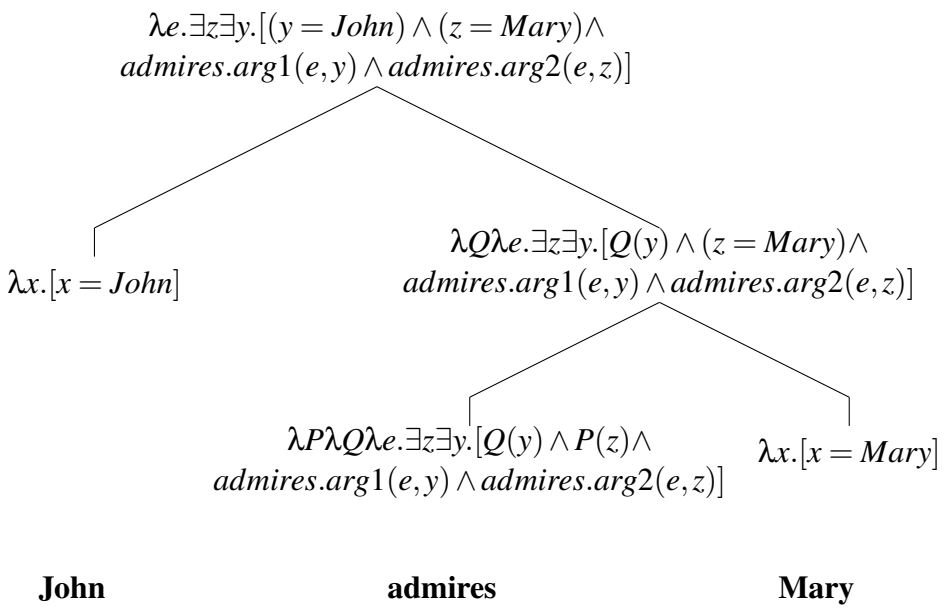

Figure 3.5: Semantic derivation for "John admires Mary". *arg*1 and *arg*2 correspond to the thematic roles *agent* and *patient*, respectively.

makes use of another function `applyRule()`, which takes two expressions and a CCG combinatory rule as arguments. It corresponds to *applyRule* in algorithm 1. `applyRule()` governs the application of functional operations between two subexpressions. That is, it maps the CCG combinatory operator onto its corresponding functional operator (e.g. $B_{>} \mapsto \circ$) and applies this functional operator to the two expressions passed in as arguments, returning the resulting composed expression.

Regarding `applyRule()`, it was noted in section 3.1.1 that function composition, substitution, and type-raising are not implemented in the `nltk.sem.logic` module. These three functions were implemented as part of `SemanticComposer` with the plan of migrating them into `nltk.sem.logic` at a later time. Thus `applyRule()` maps CCG application ($>$ or $<$) to function application implemented in the `logic` module, and the other CCG operators to their corresponding implementations in the `semparse` package. Table 3.1 outlines this mapping.

`buildExpression()` makes use of two other classes: `SyntacticCategory` and `SemanticCategory`. These are used for building the logical expressions for the tokens in the input sentence. Together, the instantiations of `SyntacticCategory` and `SemanticCategory` correspond to line 3 of algorithm 1, i.e. the *semanticCategory*() function. When `buildExpression()` reaches a leaf node, first `SyntacticCategory` is instantiated, which holds information about the CCG category and indexed category. This is then passed to the constructor for `SemanticCategory`, which determines the

CCG operation	Function call
Forward application ($>$)	`ApplicationExpression(left, right)`
Backward application ($<$)	`ApplicationExpression(right, left)`
Forward composition ($B_>$)	`compose(left, right)`
Backward composition ($B_<$)	`compose(right, left)`
Forward substitution ($S_>$)	`substitute(left, right)`
Backward substitution ($S_<$)	`substitute(right, left)`
Type-raising (T)	`typeraise(expr)`

Table 3.1: Mappings from CCG operation to function operation as used by `applyRule()`. `left` and `right` refer to the left and right child expressions, respectively.

logical expression for the token at that leaf node. The next two sections describe these classes in greater detail.

3.2.2 Syntactic Category Implementation

The `SyntacticCategory` class represents a CCG syntactic category. The class diagram is shown in figure 3.6.

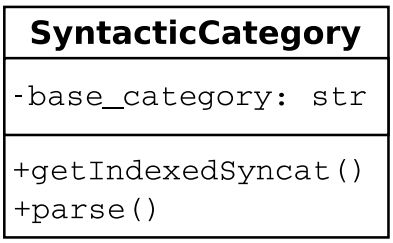

Figure 3.6: Class diagram for `SyntacticCategory`.

The `base_category` attribute is the CCG syntactic category assigned by the CCG lexicon, e.g. $(S \setminus NP)/NP$. It is passed as a string to the class constructor. The `getIndexedSyncat()` method translates `base_category` into its indexed counterpart, e.g. $(S \setminus NP)/NP \mapsto (S_e \setminus NP_z)/NP_y$. This is accomplished via a file provided by the C&C CCG parser which defines these mappings [Clark and Curran, 2004]. This file is included in the `semparse` package under `data/lib/markedup`. The `SyntacticCategory.parse()` method parses the indexed syntactic category into the binary tree representation of the function. This is done using a simple recursive-descent parser implementation. The rules used for parsing the indexed syntactic category are

provided below.

Rules for the recursive-descent parser

$Expr \leftarrow Term\ \ Op\ \ Term$

$Term \leftarrow PrimitiveCategory$

$Term \leftarrow (Expr)$

$Op \leftarrow \backslash$

$Op \leftarrow /$

Where *PrimitiveCategory* is a CCG primitive category, e.g. NP_x.

The return value of `SyntacticCategory.parse()` is a nested Python list representing the tree. Thus, taken together, the two methods of the `SyntacticCategory` class are the implementation of the first step in building the functional definition of the semantic category as outlined in section 2.3.2.1, that is, translating the syntactic category into an indexed syntactic category and parsing the indexed category into a binary tree representing the function.

3.2.3 Semantic Category Implementation

The `SemanticCategory` class governs the creation of a semantic expression for a given token in the input. It holds an instance of `SyntacticCategory`. The diagram for this class is given in figure 3.7.

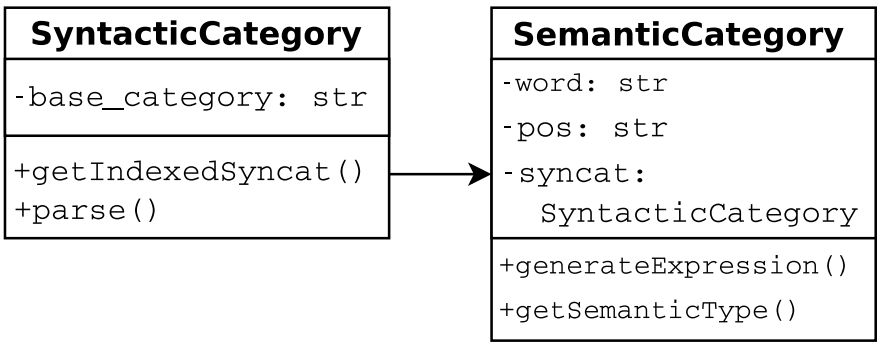

Figure 3.7: Class diagram for `SemanticCategory` showing how it interacts with `SyntacticCategory`.

The constructor for `SemanticCategory` takes three arguments: word, POS tag, and a syntactic category. These are used by the `getSemanticType()` and

`generateExpression()` methods to determine the semantic expression for the token.

The `syncat` attribute is an instance of `SyntacticCategory`. Thus, to instantiate `SemanticCategory` it is first necessary to instantiate `SyntacticCategory`. The `generateExpression()` and `getSemanticType()` methods of the `SemanticCategory` class carry out steps 2 through 4 of the functional definition construction process outlined in section 2.3.2.1 (variable assignment, predicate-argument variable pairing, construction of the logical expression), as well as the construction of the semantic definition of the semantic category (section 2.3.2.2) These methods are outlined below.

3.2.3.1 The `getSemanticType()` Method

This method assigns the `SemanticCategory` instance one of the semantic types outlined in table 2.1 in section 2.3.2.2. To reiterate, these semantic types are COPULA, UNIQUE, NEGATION, NOTEXISTS, TYPE, ENTITY, EVENT, TYPEMOD, EVENTMOD, COUNT, QUESTION. This assignment is done according to the lemma and POS tag rules given in table 2.1. The semantic type is used by `generateExpression()` to determine the appropriate semantic definition for the semantic category.

3.2.3.2 The `generateExpression()` Method

The input to this method is a `SyntacticCategory` object and its output is a Neo-Davidsonian expression representing the semantics of `SemanticCategory.word`. This method proceeds in the following manner:

1. The indexed syntactic category is parsed into a binary tree (`SyntacticCategory.parse()`).
2. Nodes in the tree are assigned variable names. The predicate and argument variables are paired and returned as an ordered set (`SemanticCategory.getVariables()`).
3. The functional definition is constructed according to the ordered set (`SemanticCategory.buildStem()`).
4. The semantic definition is constructed and appended to the functional definition.

`SemanticCategory.{getVariables(), buildStem()}` are not shown in the class diagram in figure 3.7 as they are not externally visible methods. `getVariables()` works by traversing the tree top-down. When it reaches a non-leaf node, it assigns that

node a function variable (e.g. P or Q) and descends that node's right-hand child (i.e. its argument) until it reaches a leaf. This leaf is then assigned an individual variable (e.g. x or y) and paired with the predicate variable. The pair is then added to an ordered set. The function also returns the variable that is finally returned by the function, e.g. e. `buildStem()` works by first determining which variables are λ variables and which are existentially quantified. This is done according to the following rules:

λ variables := $\{P|P \in predVars(tree) \cup \{indVar(tree)\}\}$
∃ variables := $\{x|x \in argVars(tree) \setminus \{indVar(tree)\}\}$

Where $predVars(tree)$ is the set of predicate variables in the tree, $indVar(tree)$ is the variable that is finally returned by the function (i.e. the bottom left-most leaf of the tree), and $argVars(tree)$ is the set of argument variables in the tree. Thus, given the example tree in figure 3.8:

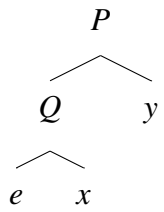

Figure 3.8: Parse tree of $(S_e \setminus NP_x)/NP_y$ assigned variable names.

$predVars(tree) \mapsto \{P,Q\}$
$indVar(tree) \mapsto e$
$argVars(tree) \mapsto \{y,x\}$
λ terms = $\{P,Q,e\}$
∃ variables = $\{y,x\}$

`buildStem()` then builds the functional definition in the following way:

1. A `logic.ApplicationExpression` is constructed for each predicate variable applied to its arguments (e.g $P(y)$, $Q(x)$).

2. A `logic.AndExpression` is created by conjoining the `ApplicationExpressions` (e.g. $[P(y) \wedge Q(x)]$).

3. A `logic.ExistsExpression` is created from the `AndExpression` by adding the ∃ variables (e.g. $\exists y \exists x.[P(y) \wedge Q(x)]$).

4. A `logic.LambdaExpression` is created from the `ExistsExpression` by adding the λ terms (e.g. $\lambda P \lambda Q \lambda e.\exists y \exists x.[P(y) \wedge Q(x)]$).

This process ensures that the expression at each step is a valid `logic.Expression` object and that its associated methods can be employed for constructing the expression in the subsequent steps.

The next step in `generateExpression()` is to construct the semantic definition. As mentioned in section 2.3.2.2, the semantic type alongside the structure of the functional definition determines the form of the semantic definition. In practice, each semantic type is mapped to a function which takes a functional definition as an argument, determines the semantic definition, and returns the full expression (functional definition + semantic definition). These functions are implemented in `nltk.semparse.rules`. In general, these rule functions follow the process below:

1. Construct one or more subexpressions according to the semantic type and argument variables in the functional definition
   (e.g. EVENT $\Rightarrow \{admire.arg1(e,x), admire.arg2(e,y)\}$).

2. Add these subexpressions to the functional definition
   (e.g.$\lambda P \lambda Q \lambda e.\exists y \exists x.[P(y) \wedge Q(x) \wedge admire.arg1(e,x) \wedge admire.arg2(e,y)]$).

Where $arg1$ and $arg2$ correspond to the Neo-Davidsonian thematic roles *agent* and *patient*, respectively. The associations between semantic type and predicates to be used in the semantic definition are given in table 2.1.

3.2.3.3 Special Cases

There are cases in which `generateExpression()` as described above does not produce a correct semantic representation for a given token. These are language-specific phenomena that must be dealt with case by case. This section describes how these phenomena are handled.

For example, consider the verb "is" functioning as copula. In this case "is" has the indexed syntactic category $(S_e \setminus NP_x)/NP_y$. Following the above procedure, the semantic category for "is" would be $\lambda P \lambda Q \lambda e.\exists y \exists x.(P(y) \wedge Q(x) \wedge is.arg1(e,x) \wedge is.arg2(e,y))$. This form does not, however, correctly capture the semantics of the copula. The copula is used to link the subject with a predicate rather than, as with transitive verbs, to describe an action of a subject upon an object. Thus, for the sentence "John is smart", the

semantic representation should be $smart(x) \wedge (x = John)$ rather than $smart(y) \wedge (x = John) \wedge is.arg1(e,x) \wedge is.arg2(e,y)$. It is necessary, then, to treat "is" functioning as a copula as a special case and assign it the semantic category $\lambda P \lambda Q.\exists y \exists z.[(\lambda x.[x = y])(z) \wedge P(z) \wedge Q(y)]$. Using this, y and z will be equated and, in the "John is smart" example, the composed expression will be $smart(x) \wedge (x = John)$.

Special cases are described in the configuration file `data/lib/specialcases.txt`. This file was built by hand drawing largely from the file of the same name in GRAPHPARSER. The file has four tab separated fields: lemma, POS tag, indexed syntactic category, and semantic type/semantic category. A token is a special case if its lemma, POS tag, and indexed syntactic category match the first three fields of some entry in the configuration file. The fourth field can be either one of the semantic types given in section 3.2.3.1 or a logical expression. If it is a semantic type, that semantic type will be assigned to the `SemanticCategory` instance and `generateExpression()` will proceed as described above. If it is a logical expression, `generateExpression()` is not called and the logical expression will be assigned to the `SemanticCategory` instance directly.

In practice, the lemma and POS tag fields are regular expressions, e.g. `not|n\'t` for the lemma field or `JJ.?` for the POS tag field. The indexed syntactic category is a string literal that must be matched exactly, e.g. `((S{_}\NP{Y}<1>){_}/NP{Z}<2>){_}`, which is the ascii representation of $(S_e \setminus NP_y)/NP_z$

3.3 Usage

There are two steps to using the semantic parser: 1. instantiate the `SemanticParser` class with a CCG lexicon; 2. parse a sentence using the `SemanticParser.parse()` method.

The instantiation of `SemanticParser` requires a CCG lexicon. The CCG lexicon must be an instance of `nltk.ccg.CCGLexicon`. See section 3.1.2 for details on defining a CCG lexicon with the `nltk.ccg` package.

The `SemanticParser.parse()` method requires a tokenized and POS tagged sentence as input. This method yields a `nltk.semparse.Derivation` object for each syntactic parse of the input sentence. Tokenization and POS tagging may be carried out using the `nltk.word_tokenize` and `nltk.pos_tag` functions or other programs, but the input must be of the format of the return value of `nltk.pos_tag`. This format is a Python list of tuples where the first member is the word and the second

is the POS tag. For example, the correct format for the sentence "I eat peaches" is `[('I', 'PRP'), ('eat', 'VBP'), ('peaches', 'NNS')]`. Note that `SemanticParser.parse()` has a configuration parameter `n`, which limits the number of parses to yield. The default option is `n=0`, which yields all possible parses. A usage example of the semantic parser is shown below:

```
>>> from nltk import word_tokenize, pos_tag
>>> from nltk.ccg import lexicon
>>> from nltk.semparse import SemanticParser
>>> ccglex = lexicon.parseLexicon(r'''
...     :- S, N
...     I => N
...     eat => (S\N)/N
...     peaches => N
... ''')
>>> semparser = SemanticParser(ccglex)
>>> sent = "I eat peaches."
>>> tagged_sent = pos_tag(word_tokenize(sent))
>>> for parse in semparser.parse(tagged_sent, n=5):
...     print parse.getExpression()
```

3.4 Summary

This chapter discussed the details of implementing the semantic parser design outlined in chapter 2. Two existing NLTK packages were used in the implementation: `nltk.sem.logic` for working with logical expressions and `nltk.ccg` for CCG parsing. The semantic parser is implemented in the `nltk.semparse` package. Its main class is `SemanticParser`. This class uses instances of `nltk.ccg.CCGChartParser` for syntactic parsing and `nltk.semparse.SemanticComposer` for composing logical expressions. `SemanticComposer` uses the classes `SyntacticCategory` and `SemanticCategory` for representing CCG categories and building logical expressions for tokens, respectively. Language-specific phenomena that result in `SemanticCategory` failing to generate a semantically correct logical expression for a token are handled by the configuration file `data/lib/specialcases.txt`.

Chapter 4

Evaluation

The semantic parser was evaluated against the GEOQUERY880 dataset. GEOQUERY is a database of geographical facts about the U.S. and is widely used in the evaluation of semantic parsers. The dataset consists of 880 questions about geography paired with gold-standard MRs. These MRs are, however, in the `geo-funql` query language, which is an MRL engineered specifically for this dataset. The gold-standard MRs are, then, not directly comparable to the output of the `nltk.semparse` system. Thus it is not possible to evaluate the correctness of the system in terms of accuracy. It is, however, possible to calculate the coverage, that is, the number of questions for which the system outputs a logical expression that covers the entire input. The correctness of the parser is evaluated with a qualitative analysis.

Section 4.1 discusses the coverage of both the syntactic and semantic components of the system. Section 4.2 contains the qualitative analysis of the output of the semantic parser for three example questions from the GEOQUERY880 dataset. This analysis is comprised of separate analyses of the syntactic and semantic derivations as well as a comparison of the output to that of the system implemented by [Reddy et al., 2014], upon which this system was modelled.

4.1 Coverage

The CCG lexicon used was built specifically for this dataset to maximise the coverage of the syntactic component. It was built using the output of the C&C supertagger and formatted to adhere to the requirements of the `nltk.ccg.CCGLexicon` class. The questions were parsed using the `nltk.semparse` system and the coverage of both the syntactic and semantic components was calculated. Table 4.1 shows the coverage of

the system on the GEOQUERY880 dataset.

Parse	Questions Parsed	Coverage
Syntax	825	93.75%
Semantics	366	41.59%
Total	**880**	

Table 4.1: Coverage of syntactic and semantic components of `nltk.semparse` on the GEOQUERY880 dataset.

The coverage of the syntactic component is not 100% for two reasons: 1) unlike the C&C parser, `nltk.ccg` does not allow unary rules such as $N \rightarrow NP$. This means that certain syntactic categories will be unable to combine as would be possible under the C&C parser. 2) the C&C parser is able to use non-standard combinatory rules in parsing, while `nltk.ccg` is not. The supertagger assigns syntactic categories accordingly, so some of the syntactic categories in the CCG lexicon will not be able to combine according to the rules possible under `nltk.ccg`.

The following limit the coverage of the semantic component:

Tokenizer limitation

The current implementation does not have any special handling of compound nouns, e.g. "South Dakota". The tokenizer used (`nltk.word_tokenize`) treats each token of the compound as separate. The CCG lexicon assigns them separate syntactic categories, but semantically they should be treated as a single entity. In the "South Dakota" example, both "South" and "Dakota" are given the semantic category for proper nouns (e.g. $\lambda x.[x = South]$) and they are not able to combine in the semantic derivation.

POS tagger limitation

The default NLTK POS tagger has a token accuracy of 59.75% when evaluated against the Brown corpus [1]. This is considerably lower than the state-of-the-art, which is around 97% for the Stanford tagger [Manning, 2011]. This indicates that a number of failures in the semantic derivation are caused by incorrect POS tags, which likely result in incorrect semantic category assignments. Indeed, `nltk.pos_tag` fails in every case to assign the correct POS tag to the verb "border", as in "What states border Texas?". It is always assigned NN or NNS. This means that the word "border" and its

[1] `nltk.pos_tag` was evaluated using the `evaluate` method against the last 10% of the Brown corpus in NLTK (`nltk.corpus.brown`)

conjugations are assigned an incorrect semantic category making them unable to combine correctly. There are 133 sentences in the dataset that contain some conjugation of the verb "border". Removing all sentences that contain "border" increases coverage to **48.46%**.

Additionally, long-range verb dependencies often result in incorrect POS tag assignment. For example, in "What state does the longest river cross?", "cross" is assigned the POS tag NN, again resulting in an incorrect semantic category assignment.

4.2 Correctness

This section gives an analysis of three example parses. For each parse, there is an analysis of the syntax and the semantics, as well as a comparison of the output logical expression to the output of the GRAPHPARSER system developed by [Reddy et al., 2014].

What is the biggest state?
This question is parsed correctly syntactically and semantically. The syntactic parse is shown in figure 4.1.

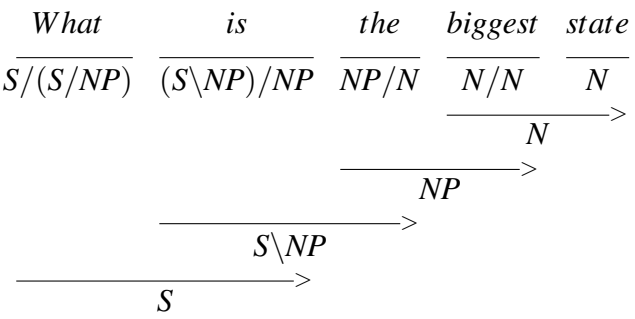

Figure 4.1: Syntactic derivation for "What is the biggest state?"

The noun-phrase "the biggest state" is derived first and then modified by "What". The composed logical expression output for this parse is

$$\exists x.[state(x) \wedge biggest(x) \wedge UNIQUE(x) \wedge TARGET(x)]$$

This logical expression correctly captures the semantics of the input question. The question is asking for one ($UNIQUE$) state that satisfies the constraint *biggest*. The output of GRAPHPARSER for this question is

$$state(s,x) \wedge state.biggest(s',x) \wedge UNIQUE(x) \wedge QUESTION(x)$$

Note that GRAPHPARSER does not use quantifiers in the output. The *QUESTION* predicate corresponds to the *TARGET* predicate in the `nltk.semparse` output. $state.biggest(s',x')$ in the GRAPHPARSER output corresponds to $biggest(x)$ in the `nltk.semparse` output. The additional argument (s and s') to the *state* predicate serves as an identifier, similar to how the event variable e serves as an identifier of an event, and can be ignored in this case. Each predicate in the GRAPHPARSER output has a corresponding predicate in the `nltk.semparse` output. The two are, then, equivalent.

"How big is Alaska?"

The semantic parser fails to compose a semantic expression for the first syntactic parse output by the `nltk.semparse`, shown in figure 4.2. Examining the syntactic parse, it is evident that the semantic interpretation of the syntactic parse is incorrect.

$$\dfrac{\dfrac{How}{S/S} \quad \dfrac{\dfrac{\dfrac{big}{NP/NP} \quad \dfrac{is}{(S\backslash NP)/NP}}{S/NP}\,{<}\mathbf{S}_{\times} \quad \dfrac{Alaska}{NP}}{S}\,{>}}{S}\,{>}$$

Figure 4.2: Syntactic derivation for "How big is Alaska?".

The problem lies with the syntactic category assigned to "How" and "big". "big" is treated as a normally functioning adjective. That is, the syntactic category NP/NP modifies the noun-phrase to its right, e.g. "big state". The category NP/NP allows "big" to combine with "is" by backward substitution. Further combining with "Alaska", the result is a factual statement meaning *Alaska is big*. This combines in the last step with "How" to obtain the start symbol S. The final semantic interpretation of this parse is then *Alaska is big, how so?* or *How is it that Alaska is big?*. Thus in this parse "How" modifies the rest of the sentence when it should modify only "big".

The correct syntactic derivation is shown in figure 4.3. Here, "How" modifies "big" only. This is consistent with the desired semantic interpretation: *To what degree is Alaska big?*. This parse results in a correct logical form:

$$\exists y \exists z \exists d.[(y = z) \wedge (z = \text{alaska}) \wedge big(d,y) \wedge TARGET(d)]$$

$$\dfrac{\dfrac{\dfrac{\overline{How}}{(S/(S/(S\backslash NP)))/(S\backslash NP)} \quad \dfrac{\overline{big}}{S\backslash NP}}{S/(S/(S\backslash NP))}> \quad \dfrac{\dfrac{\overline{is}}{(S/(S\backslash NP))/NP} \quad \dfrac{\overline{Alaska}}{NP}}{S/(S\backslash NP)}>}{S}>$$

Figure 4.3: Syntactic derivation for "How big is Alaska?"

The equality expressions mean that, e.g. y in every case can be replaced with z. Simplifying the equality expressions the expression becomes

$$\exists d.[big(d, \text{alaska}) \wedge TARGET(d)].$$

The semantics of this expression are the following: Alaska is big to some degree d, and the object of the question is that degree d. GRAPHPARSER produces the expression:

$$big(e,x) \wedge is.arg1(e, \text{alaska}) \wedge is.arg2(e,x) \wedge QUESTION(x)$$

`nltk.semparse` treats "is" as a copula, while GRAPHPARSER does not. This results in a semantically unclear expression from `GraphParser`, since it is not clear that the object of the question (x) is the degree to which Alaska is big.

What state is Dallas in?

This question is syntactically parsed, but no parse output by the syntactic component results in a full logical expression. The syntactic derivation for this question is shown in figure 4.4.

The semantic derivation proceeds correctly through the first three steps (type-raising "Dallas" and the first two forward composition operations). At this point in the derivation, after simplifying the equality, the expressions is

$$\lambda P \lambda z.\exists e.[P(z) \wedge in.arg1(e, dallas) \wedge in.arg2(e,z)]$$

The semantics of this partial expression is: *Dallas is in some yet undefined z.* The semantic derivation fails during the backwards substitution operation. This operation

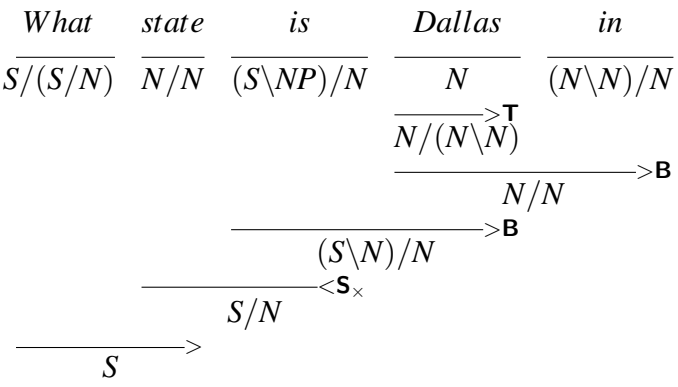

Figure 4.4: Syntactic derivation for “What state is Dallas in?”.

passes the predicate for “state” $\lambda x.[state(x)]$ in as the second argument (z) to the partial expression resulting in

$$\lambda P.\exists e.[P(\lambda x.[state(x)]) \wedge in.arg1(e,dallas) \wedge in.arg2(e,\lambda x.[state(x)])]$$

The correct operation here would be backward functional application. This would pass $\lambda x.[state(x)]$ in to the predicate argument P of the partial expression, resulting in

$$\lambda z.\exists e.[state(z) \wedge in.arg1(e,dallas) \wedge in.arg2(e,z)]$$

which could then combine with “What” to form

$$\lambda z.\exists e.[state(z) \wedge in.arg1(e,dallas) \wedge in.arg2(e,z) \wedge TARGET(z)]$$

GRAPHPARSER successfully composes the following logical expression for the question:

$$state(s,x) \wedge is.arg2(e,dallas) \wedge is.in.arg2(e,x) \wedge QUESTION(x)$$

Again here GRAPHPARSER treats “is” as a non-copular verb, while `nltk.semparse` treats it as copular. GRAPHPARSER is correct in this case, as “is” does not link the subject to a predicating element. This expression has one issue: there is no first argument to the *is* predicate. Instead, “Dallas” is treated as the second argument to *is*, when it should be the first.

Chapter 5

Conclusion

This work presented a system for determining wide-coverage semantic representations from CCG parses within the NLTK framework. The system was based on the GRAPHPARSER semantic parser developed by Reddy et. al. 2014, which parses NL input into ungrounded logical forms and then maps these onto FREEBASE graphs. The `nltk.semparse` system was designed with two main goals: practicality and transparency. The practicality goal was addressed by the fact that the system outputs ungrounded logical forms. It is possible to learn a system which maps these ungrounded forms to a target query language. The transparency goal was addressed by the use of CCG, which has a transparent interface between syntax and semantics.

Chapter 1 provided the background and motivation for the project. Semantic parsers suffer from a limitation in coverage due to a lack of annotated training data from which to learn the mapping from NL to MR and datasets/ontologies upon which to ground the MRs. Recent work in scaling semantic parsers has focused on both reducing the reliance on annotated training data and making the output MRs able to be grounded in more domains. The GRAPHPARSER semantic parser addresses the training data issue by developing a process for translating indexed CCG categories into logical forms. Since these logical forms are ungrounded but expressive (using Neo-Davidsonian event semantics), a mapping is learnt from the logical forms to a target MR, in this case a FREEBASE graph, addressing the grounding issue. Furthermore, the use of CCG provides a transparent interface between syntax and semantics.

Chapter 2 outlined the design of the `nltk.semparse` system. The system is broken into two components: a syntactic component, which is comprised of a CCG parser and a CCG lexicon, and a semantic component, which is comprised of a lexical semantics generator and a semantic composer. The lexical semantics generator uses the indexed

CCG category to determine a Neo-Davidsonian logical expression (semantic category) for a given token. It does this by analyzing the category as a function and translating this function into a λ expression. The semantic composer then composes the semantic categories for the tokens in the input sentence according to the CCG operators to obtain a semantic representation of the full input. Chapter 3 detailed the implementation of this design.

Chapter 4 presented the coverage of `nltk.semparse` on the GEOQUERY880 dataset. It also provided a qualitative analysis of the correctness of the parser's output logical forms.

5.1 Future Work

The current implementation could be improved upon in the following ways:

- As discussed in section 4.1, compound nouns are not treated as single entities, which results in a failure to compose full logical expressions for sentences such as "How big is South Dakota?". Further preprocessing could be done to identify compound nouns and treat them as single entities, for example with named-entity recognition (NER).

- The CCG parser is not probabilistic and thus the parses it outputs are not ranked in any way. From a practical usage perspective, it would be ideal to have more likely parses evaluated first so that the system could output a semantics more quickly. Further work could be done to make `nltk.ccg` probabilistic.

- Information from sources such as VerbNet or WordNet could be used to refine the semantic type and semantic definitions determined by the lexical semantics generator. Information such as number and type of arguments to verb predicates could result in more accurate semantic definitions.

- It would be beneficial to develop a system for learning mappings from the ungrounded semantic representations to grounded MRs. For example, a mapping to `geo-funql`. This would provide possibilities for more meaningful evaluation. It would be possible to measure, for example, the accuracy of the system with regards to querying the GEOQUERY database. Building off of this, it would be beneficial to implement an API to the GEOBASE query engine in NLTK for the purposes of demonstration and evaluation.

Appendices

.1 A Special Case of Semantic Category Generation

This appendix discusses the generation of semantic categories for more complex examples. Consider the generation of the functional definition of the semantic category for "not" :: $(S_e \setminus NP_z)/(S_e \setminus NP_z)$. First, the indexed syntactic category is parsed into a binary tree where the right child is the argument and left child is the return value:

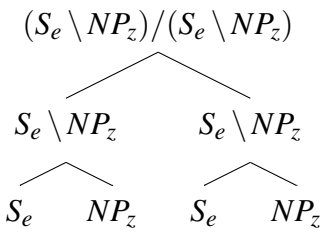

Figure 1: Parse of the indexed syntactic category $(S_e \setminus NP_z)/(S_e \setminus NP_z)$ for the word "not".

Then variable names are assigned:

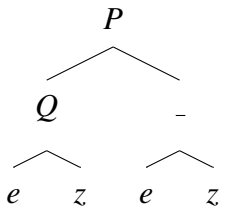

Figure 2: Parse tree from figure 2.3 assigned variable names.

In contrast to the example given in section 2.3.2, here one function takes another function as its argument. Namely, P takes a function which takes z as its argument and returns e. Note that P's argument and return value, Q, have the same argument variable, i.e. z. This can also be seen in the indexed syntactic category, as both NP's have the same subscript variable and thus the same referent. In building the functional definition from this syntactic category we must, then, ensure that the argument function to P and the function returned from P operate over the same variable. This is done by making the first argument to P in the functional definition an equality expression.

$$\lambda P \lambda Q \lambda e. \exists z. [P(\lambda x. (z = x))(e) \wedge Q(z)]$$

The semantic definition is then added, forming the full semantic category for "not", $\lambda P \lambda Q \lambda e. \exists z. [P(\lambda x. [z = x])(e) \wedge Q(z) \wedge NEGATE(e)]$. The parse in figure 3 illustrates

how this equality expression functions when combined with the subexpression for a partial verb phrase.

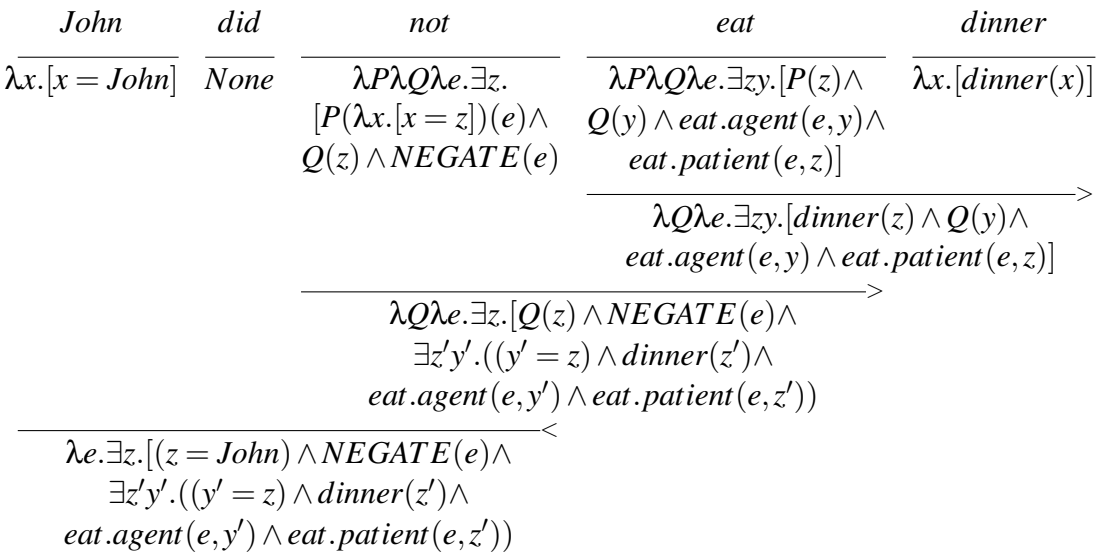

Figure 3: Semantic derivation illustrating the function of the equality subexpression in the semantic category for “not”.

Here, z must be equated to y' because they both denote the same entity, namely “John”. This is done by the equality subexpressions of the semantic category for “not”.

Bibliography

[Androutsopoulos et al., 1995] Androutsopoulos, I., Ritchie, G. D., and Thanisch, P. (1995). Natural language interfaces to databases - an introduction. *CoRR*, cmp-lg/9503016.

[Bird et al., 2009] Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media.

[Clark and Curran, 2004] Clark, S. and Curran, J. R. (2004). Parsing the wsj using ccg and log-linear models. In *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Clarke et al., 2010] Clarke, J., Goldwasser, D., Chang, M.-W., and Roth, D. (2010). Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, CoNLL '10, pages 18–27, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Ge and Mooney, 2005] Ge, R. and Mooney, R. J. (2005). A statistical semantic parser that integrates syntax and semantics. In *Proceedings of CoNLL-2005*, Ann Arbor, Michigan.

[Goldwasser et al., 2011] Goldwasser, D., Reichart, R., Clarke, J., and Roth, D. (2011). Confidence driven unsupervised semantic parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 1486–1495, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Kate and Mooney, 2006] Kate, R. J. and Mooney, R. J. (2006). Using string-kernels for learning semantic parsers. In *In Proc. of COLING/ACL-06*, pages 913–920.

[Kwiatkowski et al., 2013] Kwiatkowski, T., Choi, E., Artzi, Y., and Zettlemoyer, L. (2013). Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1545–1556, Seattle, Washington, USA. Association for Computational Linguistics.

[Manning, 2011] Manning, C. D. (2011). Part-of-speech tagging from 97% to 100%: is it time for some linguistics? *Computational Linguistics and Intelligent Text Processing*, pages 171–189.

[Miller et al., 1996] Miller, S., Stallard, D., Bobrow, R., and Schwartz, R. (1996). A fully statistical approach to natural language interfaces. pages 55–61.

[Parsons, 1990] Parsons, T. (1990). *Events in the Semantics of English*. MIT Press, Cambridge, MA.

[Poon and Domingos, 2009] Poon, H. and Domingos, P. (2009). Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*, EMNLP '09, pages 1–10, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Reddy et al., 2014] Reddy, S., Lapata, M., and Steedman, M. (2014). Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics (TACL)*.

[Steedman, 1996] Steedman, M. (1996). A very short introduction to ccg. `http://www.inf.ed.ac.uk/teaching/courses/nlg/readings/ccgintro.pdf`. Retrieved: 2015-08-12.

[Steedman, 2000] Steedman, M. (2000). *The Syntactic Process*. MIT Press, Cambridge, MA, USA.

[Steedman and Baldridge, 2011] Steedman, M. and Baldridge, J. (2011). *Combinatory Categorical Grammar*, pages 181–224. Wiley-Blackwell.

[Titov and Klementiev, 2011] Titov, I. and Klementiev, A. (2011). A bayesian model for unsupervised semantic parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 1445–1455, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Wong and Mooney, 2006] Wong, Y. W. and Mooney, R. J. (2006). Learning for semantic parsing with statistical machine translation. In *Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, HLT-NAACL '06, pages 439–446, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Zelle and Mooney, 1996] Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *AAAI/IAAI*, pages 1050–1055, Portland, OR. AAAI Press/MIT Press.