Combinatory Categorial Grammar Parser in Natural Language Toolkit

Tanin Na Nakorn

Master of Science

European Master in Informatics

School of Informatics

University of Edinburgh

2009

Abstract

In this project, a Combinatory Categorial Grammar (CCG) parser has been implemented in the Natural Language Toolkit (www.nltk.org). It is based on the existing CCG parser in NLTK implemented by Graeme Gange. As Gange's parser has many shortcomings, the parser in this project extends it to support feature and feature unification, semantic derivation, and probabilistic parsing. The parser has also improved the packed chart technique, which outperforms Gange's parser, as shown in the evaluation. The parser is also evaluated against CCGBank for its correctness.

Perfection is unattainable;
improvement is always possible.
- Ewan Klein

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.

(Tanin Na Nakorn)

Table of Contents

1 Introduction

Natural Language Processing (NLP) is a field in both Informatics and Linguistics which deals with how computer processes natural languages. The ultimate goal of NLP is to enable machines to understand a natural language and manipulate it. Currently, NLP researchers are nowhere near the goal as NLP is an AI-complete problem, which requires extensive knowledge about the external world. On the top of it, the definition of "understanding" itself is still in debate.

NLP can be divided into many sub-problems, which are perceived as different research areas in NLP. Some major research areas in NLP are Machine Translation, Information Extraction, Information Retrieval, Natural Language Generation, Natural Language Understanding, Speech Recognition, and Question Answering.

Parsing is an NLP task, which is central to many other NLP tasks. Parsing is the process of computing valid structures of a string, or a list of tokens, according a given grammar. Parsing a sentence with a given natural language grammar is not simple, as a natural language grammar is often ambiguous. This means that one sentence might have many valid structures given a natural language grammar. It is not surprising that parsing a sentence with a given natural grammar is often perceived as an NP-complete problem.

One of the prominent grammars used for formalizing natural languages is Combinatory Categorial Grammar (CCG). CCG has been argued to be the formalism used by human (Steedman & Baldridge, 2007) because it can account for the incremental language interpretation behaviour and provides a natural linkage between syntactic structure and semantic representation. Moreover, CCG offers higher flexibility compared to other grammars; it can derive the structure for any part of a sentence without deriving the structure for the whole sentence.

In this project, a CCG parser is implemented as a module of Natural Language Toolkit (http://www.nltk.org) (Bird, Klein, & Loper, 2009). NLTK is open-source

software, written in Python, which supports various NLP tasks: POS-tagging, parsing, semantic interpretation, word-tagging, and etc. NLTK is widely adopted and has been used as a teaching tool by more than 50 universities. Due to the prominence of CCG and the ubiquity of NLTK's use, it is desirable that NLTK has a CCG parser.

In this paper, the chapters are organized as follows: Chapter 2 presents some background. The Combinatory Categorial Grammar (CCG) is introduced. Its characteristics are discussed. Then, the λ-calculus (lambda calculus) is explained as it is used for semantic representation. The present CCG parsers are also discussed. Lastly, CCGBank (Hockenmaier & Steedman, 2007), the biggest CCG corpus, is introduced.

Chapter 3 describes the problem, the purpose of the parser, the design, and the implementation of the parser. Finally, the unresolved issues are explained.

Chapter 4 describes the evaluation of the parser. The sentences from CCGBank are also used to evaluate the parser. The compatibility of the parser with CCGBank is also shown in this chapter. Moreover, the performance of the parser is also evaluated.

Chapter 5 concludes the project and possible future works are described.

2 Background

In this chapter, the background knowledge essential for building and using a CCG parser is explained. The Combinatory Categorial Grammar, the λ-calculus, which is used for semantic representation, and the chart parsing famously used for CCG are explained. How to write a semantic expression that is compatible with a given category, either primitive or complex, is also explained. Later in this chapter, three existing parsers, C&C CCG parser, OpenCCG, and NLTK CCG Parser, are also discussed. Lastly, CCGBank is introduced.

2.1 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) (Hockenmaier & Steedman, 2007) belongs to the family of categorial grammars. A categorial grammar, said to be lexicalized, represents words with categories. The information about the structure is encoded in categories, unlike a context-free grammar which explicitly defines the information about the structure as a set of rules. While a context-free grammar needs to specify rules like:

```
S → NP VP
VP → V NP
```

A categorial grammar does not need those explicit rules; instead the rules are encoded in categories. A category acts as a function, which takes an argument of certain type from either left or right depending on its directionality and produces a resulting category. It can be seen as a function mapping from one category to another category. In a categorial grammar, there are two types of categories:

- **Primitive Categories:** the elementary categories are defined based on each language. For example, English might has *S*, *NP* and *N* as its primitive categories.
- **Complex Categories:** a complex category is a combination of two categories with directionality, which is either forward or backward.

In many categorial grammars, including CCG, slash and backslash are used for representing directionality. *X/Y* is the category which takes *Y* as an argument on its right. Therefore, *X/Y Y* results in the category *X*. *X\Y* is the category which takes *Y* as an argument on its left. Therefore, *Y X\Y* results in the category *X*. The two applications, with semantic derivation, in a categorial grammar are summarized below:

Forward Application:

$X/Y : f \quad Y : a \quad \rightarrow \quad X : f(a)$

Backward Application:

$Y : a \quad X\backslash Y : f \quad \rightarrow \quad X : f(a)$

The semantic derivations are also shown above. For both the left and right function, *f* is applied with *a* in order to obtain *f(a).*

CCG is different from other categorial grammars in the way that it also has a set of combinatory rules defined. Two categories can combine to make a single category in accordance to one of the combinatory rules. Please note that the λ-calculus is adopted as the notion for semantic representation. The set of combinatory rules are shown below:

Forward Composition:

$X/Y : \lambda y.F(y) \quad Y/Z : \lambda z.G(z) \quad \rightarrow \quad X/Z : \lambda z.F(G(z))$

Forward Crossing Composition:

$X/Y : \lambda y.F(y) \quad Y\backslash Z : \lambda z.G(z) \quad \rightarrow \quad X\backslash Z : \lambda z.F(G(z))$

Backward Composition:

$Y\backslash Z : \lambda z.G(z) \quad X\backslash Y : \lambda y.F(y) \quad \rightarrow \quad X\backslash Z : \lambda z.F(G(z))$

Backward Crossing Composition:

$Y/Z : \lambda z.G(z) \quad X\backslash Y : \lambda y.F(y) \quad \rightarrow \quad X/Z : \lambda z.F(G(z))$

Forward Substitution:

$(X/Y)/Z : \lambda z\, y.F(z,y) \quad Y/Z : \lambda z.G(z) \quad \rightarrow \quad X/Z : \lambda z.F(z,G(z))$

Forward Crossing Substitution:

$(X/Y)\backslash Z : \lambda z\, y.F(z,y) \quad Y\backslash Z : \lambda z.G(z) \quad \rightarrow \quad X\backslash Z : \lambda z.F(z,G(z))$

Backward Substitution:

$Y\backslash Z : \lambda z.G(z) \quad (X\backslash Y)\backslash Z : \lambda z\, y.F(z,y) \quad \rightarrow \quad X\backslash Z : \lambda z.F(z,G(z))$

Backward Crossing Substitution:

$Y/Z : \lambda z.G(z) \quad (X\backslash Y)/Z : \lambda z\, y.F(z,y) \quad \rightarrow \quad X/Z : \lambda z.F(z,G(z))$

CCG also allows one category to be "type-raised" into another category:

Forward type-raising:

$X : x \quad \rightarrow \quad T/(T\backslash X) : \lambda f.f(x)$

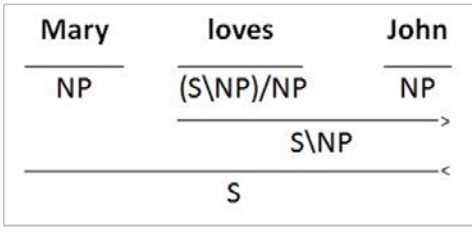
Backward type-raising:

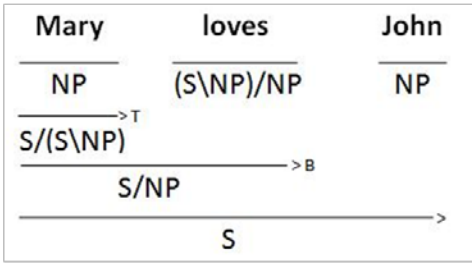
$X : x \quad \rightarrow \quad T\backslash(T/X) : \lambda f.f(x)$

With the type-raising, composition, substitution rules, CCG offers higher flexibility compared to other categorial grammars and context-free grammars. In CCG, any part of a sentence can be parsed without regard to other parts. The example of parsing the sentence "Mary loves John" is shown as follows. The lexicon is defined as shown below:

```
Mary → NP
loves → (S\NP)/NP
John → NP
```

With the lexicon defined above, there are two different parses, which are shown below:





In the example, the part “Mary loves” can be parsed to make the category *S/(S\NP)* without parsing the whole sentence. This entails that the semantic expression for “Mary loves” can be computed without regard to other parts of the sentence. However, CCG's flexibility comes with a big drawback, the famous Spurious Ambiguity. Spurious ambiguity often arises in CCG. Spurious ambiguity is the situation where a grammatical sentence has many valid semantic-equivalent parses. This causes a combinatory explosion of the search space while parsing. As any part of a grammatical sentence can form one single category, there can be many parses with the same category tagging pattern. An example where Spurious Ambiguity arises is shown below:

The	weird	beautiful	woman
NP/N : λx.the´(x)	*N/N : λx.weird´(x)*	*N/N : λx.beautiful´(x)*	*N : woman´*

According to the combinatory rules in CCG,

NP/N N/N → NP/N
N/N N/N → N/N
N → N\(N/N) and N/N N\(N/N) → N
N → NP\(NP/N) and NP/N NP\(NP/N) → NP

Therefore, the sentence has, in total, 9 semantic-equivalent parses, which are shown below:

[the [weird [beautiful woman]$_{N}$]$_{N}$]$_{NP}$
[the [weird [beautiful [woman]$_{N\(N/N)]N}$]$_{N}$]$_{NP}$
[[the weird]$_{NP/N}$ [beautiful woman]$_{N}$]$_{NP}$
[[the weird]$_{NP/N}$ [beautiful [woman]$_{N\(N/N)}$]$_{N}$]$_{NP}$
[[the weird]$_{NP/N}$ [[beautiful [woman] $_{N\(N/N)}$]$_{N}$]$_{NP\(NP/N)}$]$_{N}$
[[the weird]$_{NP/N}$ [[beautiful woman]$_{N}$]$_{NP\(NP/N)}$]$_{NP}$
[[[the weird]$_{NP/N}$ beautiful]$_{NP/N}$ woman]$_{NP}$
[the [[weird beautiful]$_{N/N}$ woman]$_{N}$]$_{NP}$
[[the [weird beautiful]$_{N/N}$]$_{NP/N}$ woman]$_{NP}$

Their semantic expressions are *the´(weird´(beautiful´(woman´)))*. The type-raising, composition, and substitution rules exponentially increase the number of valid parses. The problem becomes exponentially worse for a longer sentence.

CCG has been extended with the modalities of slash (Steedman & Baldridge, 2007; Baldridge & Kruijff, 2003). The modalities define the ability of functional categories being applied on some combinatory rules. The modalities enable the combinatory rules to use with any language. Without them, bans on some combinatory rules are required to prevent ungrammatical sentences.

Four modalities, ∗, ◊, ×, and ·, of slash are introduced. These modalities define which combinatory rules can be applied. ∗ is the most restricted type, which allows only the forward and backward application to be applied. ◊ extends ∗ by allowing the forward and backward composition and the forward and backward substitution to be applied. × extends ◊ by allowing the forward and backward crossing composition and the forward and backward crossing substitution to be applied. Lastly, · allows any combinatory rule to be applied.

2.2 The λ-calculus

The λ-calculus (lambda calculus) was first introduced by Alonzo Church in the 1930s. It is the formalism mainly used for representing computable functions. Since the semantics are often represented in a form of functions, therefore, the λ-calculus is suitable for representing semantic expressions.

The two types of the λ-calculus: the typed λ-calculus and the untyped λ-calculus. The typed λ-calculus specifies the type of a variable and the type of a function. In the typed λ-calculus, the typed of a variable applied to the function must be compatible with the type of the function. By contrast, the untyped λ-calculus does not have the mentioned restriction. In this project, the typed λ-calculus has been used. However, only two types of variables are used: functional and non-functional. A functional

variable can only be bound to a function, while a non-functional variable can only be bound to a value.

The λ-calculus consists of 3 types of expressions: variable, function, and application. A variable is an identifier which can be bound to an atom or a variable. A function can take on only one variable which is in the form *λ variable . expression*. A function with more than one variables must be represented with recursion since expression can recursively be a function. An application is represented in the form *function (expression)*. To sum up, the syntactical rules of the λ-calculus are shown below:

```
expression := variable | function | application | atom
function := λ variable . expression
application := function (expression)
```

For the convenience, a function with multiple parameters:

```
λx . (λy . (λz . (xyz)))
```

can be represented in a shorter form, which is

```
λx λy λz . xyz
```

, or, in an even shorter, which is

```
λx y z . xyz
```

Variables in the λ-calculus can be distinguished into bound variables and free variables. Bound variables are bound by the λ-operator, while free variables are not bound. In the expression,

```
λx.(xy)
```

x is said to be bound, while *y* is said to be free.

Moreover, in the λ-calculus, a function can be applied to expressions. Application of a function to expressions is the binding of the function's variables to the expressions. In the example,

```
(λx λy λz . give(x,y,z)) (john')
```

is simplified into

```
λy λz . give(john', y,z)
```

Even though the typed λ-calculus is used in this project, only the type of variables is considered. There are two types of variable: atomic variables and functional variables. In this project, an atomic variable is represented with a single lowercase letter. A functional variable is represented with a single uppercase letter.

2.3 The λ-calculus with CCG

For a category, either primitive or complex, its corresponding semantic expression takes a certain structure in order to be compatible with the category. For a primitive category, the form of its semantic expression can be in any form. However, for a complex category, the semantic expression must be a function of which the number of bound variables is equal to, or greater than, the number of the complex category's arguments and the types of those variables must be compatible with their corresponding arguments of the complex category. For example, the complex category

```
(X/Y)\Z where X, Y, and Z are primitive categories
```

has the semantic expression, which is

```
λz λy . F(z,y)
```

(X/Y)\Z consumes *Z* and then *Y*, which are primitive categories. Correspondingly, *λz y.F(z,y)* also consumes *z* and then *y*. Please note that

```
λz λy λx. F(z,y,x)
```

and

```
λz λy λX. F(z,y,X)
```

are also acceptable for the complex category above because *x* and *X* do not correspond to any argument of the complex category. Therefore, they can be of any type.

In another case, if the complex category consumes another complex category, its semantic function must consume a functional variable. For example, the complex category

```
X/(Y\Z)
```

has the semantic predicate, which is

```
λG λz . F(G(z))
```

X/(Y\Z) consumes *(Y/Z)* which is a complex category, which, in turn, consumes *Z*. Correspondingly, *λG z.F(G(z))* consumes *G* which is a function, which, in turn, consumes the atom *z*. In this project, the semantic expression defined must be compatible with its corresponding category; the parser otherwise fails or possibly returns a wrong resulting semantic expression.

2.4 Chart Parsing

Chart parsing is a suitable technique for parsing ambiguous grammars; Most of natural language grammars, such as CCG, are highly ambiguous. Chart parsing applies dynamic programming approach; The partial results are stored in a data structure called a chart. With chart parsing, a combinatory explosion of the search space is eliminated. The most popular chart parsing technique is the Cocke-Younger-Kasami (CYK) parsing (Kasami, 1965; Younger, 1967), which applies bottom-up algorithm together with dynamic programming approach. Another well-known chart parsing technique is Earley parsing (Earley, 1970), which applies top-down algorithm together with dynamic programming approach. Nonetheless, Earley parsing is not suitable for parsing CCG. The issue is explained in detail in 2.4.2

2.4.1 Cocke-Younger-Kasami (CYK) Parsing

The Cocke-Younger-Kasami (CYK) parsing algorithm is famous in its simplicity and extensibility. CYK considers every possible subsequence of the given words starting from the shortest subsequences to the longest subsequence, which contains all the words. If a subsequence can form a new non-terminal, then the non-terminal is formed and added to the chart. The computational complexity of CYK on a Chomsky-Normal-Form (CNF) grammar is $O(n^3)$. A Chomsky-Normal-Form grammar is a context-free grammar of which all production rules are in the form of:

$A \rightarrow B\ C$, or

$A \rightarrow \alpha$, or

$S \rightarrow \epsilon$

where A, B, and C are non-terminals, α is a constant, S is the start symbol and ϵ is an empty string.

Here is the pseudo-code of the algorithm:

```
Let w1,w2, …, wn be a sequence of words
Let chart be the chart
Let lex be the lexicon

for i=1 to n
 cats = lex.get(wi)
 for j=1 to sizeof(cats)
   insert cats[j] into chart[j-1,j] with no children

for span=2 to n
 for start=0 to (n-span)
  for part=1 to span
   left = start
   mid = start + part
   rend = start + span
   for A → B C in the production rule set
    if B is in chart[left,mid]
```

```
        and C is in chart[mid,rend]
      add A to chart[left,rend]
```

Since CCG is a lexicalized grammar, the CYK parsing algorithm needs some modification because CCG does not have production rules which define which category goes with which category. For example, CCG does not directly define whether or not *S* can be combined with *S\NP*. Instead, CCG have the combinatory rules that are more general. For example, the forward application rule define that *X/Y* can be combined with *Y* where *X* and *Y* can be any categories. Therefore, instead of enumerating the production rules, the parser needs to enumerate the consecutive edges and test them with each combinatory rule. With the modification, the computational complexity of CYK on CCG is $O(rn^5)$ where *r* is the number of the combinatory rule and *n* is the number of the words. Here is the pseudo-code of the algorithm:

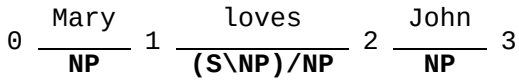
```
Let w1, w2, …, wn be a sequence of words
Let chart be the chart
Let lex be the lexicon

for i=1 to n
 cats = lex.get(wi)
 for j=1 to sizeof(cats)
   insert cats[j] into chart[j-1,j] with no children

for span=2 to n
 for start=0 to (n-span)
  for part=1 to span
   left = start
   mid = start + part
   rend = start + span

   for left in { l | l in chart[lstart,mid] }
    if left can be forward type-raised
     add the new edge to chart[lstart,mid]
    if left can be backward type-raised
```

```
    add the new edge to chart[lstart,mid]

  for right in { r | r in chart[mid,rend] }
   if right can be forward type-raised
    add the new edge to chart[mid,rend]
   if right can be backward type-raised
    add the new edge to chart[mid,rend]

  for left in { l | l in chart[lstart,mid] }
   for right in { r | r in chart[mid,rend] }
    for rule in the combinatory rule (no type-raising)
      if left and right can form a new edge according
         to rule
       add the new edge to chart[lstart,rend]
```

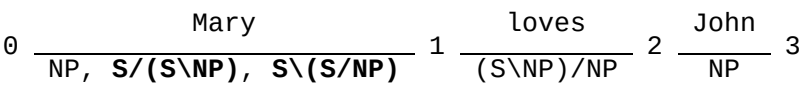
The snapshots of parsing “Mary loves John” are shown as follows.

Here we use a bracket for the set of edges which has a certain spanning. For example, *Chart[1,2]* is the set of edges that spans from the 1st position to the 2nd position. The bracket is also used for indicating the spanning of an edge. For example, *NP[2,3]* means that the edge *NP* spans from the 2nd position to the 3rd position.

First, the chart is initialized with the category of each word. The chart is:

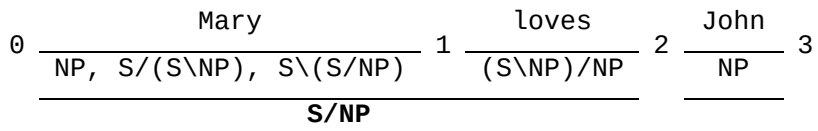


In the first iteration, where *lstart=0*, *mid=1*, and *rend=2*, the type-raised categories are added. Here the type-raising operations are allowed only on primitive categories for the sake of simplicity. *NP* is forward and backward type-raised into *S/(S\NP)* and *S\(S/NP)*. The chart becomes:

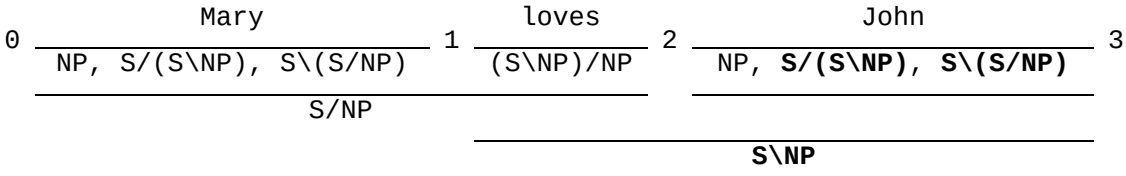


Then, the edges in [0,1] are checked with the edges in [1,2]. The new category

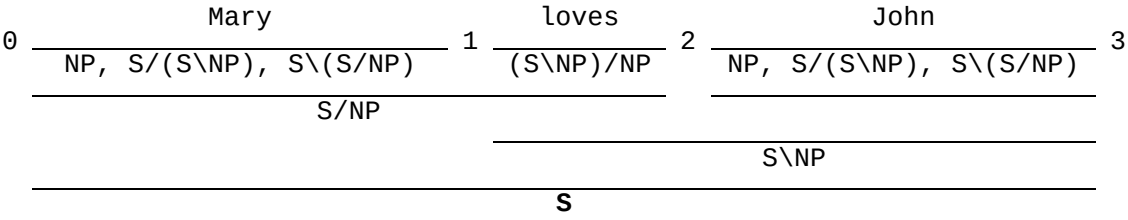
S/NP[0,1] is formed from *S/(S\NP)[0,1]* and *(S\NP)/NP[1,2]* according to the forward composition rule. The chart becomes:



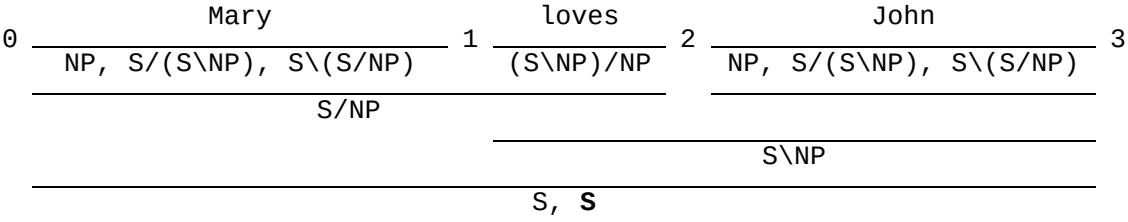
The second iteration, where *lstart=1, mid=2*, and *rend=3*, checks the edges spanning from 1 to 2 with the edges spanning from 2 to 3. *NP[2,3]* is forward and backward type-raised into *S/(S\NP)[2,3]* and *S\(S/NP)[2,3]*. *(S\NP)/NP[1,2]* can be combined with *NP[2,3]* to form *S\NP[1,3]* according to the forward application. Therefore, the chart becomes:



The third iteration checks the edges in [0,1] with the edges [1,3]. S[0,3] is formed from *NP[0,1]* and *S\NP[1,3]* according to the backward application rule. The chart becomes:



The fourth iteration checks the edges in [0,2] with the edges in [2,3]. *S[0,3]* is formed from *S/NP[0,2]* and *NP[2,3]* according to the forward application rule. The chart becomes:



In the end, there are two resulting categories: *S* and *S*. Therefore, "Mary loves John" has 2 valid parses.

2.4.2 Earley Parsing

The Earley parsing algorithm is more complicated than the CYK parsing algorithm. The Earley parser operates on a sequence of sets. Given a sequence of words w_1, w_2, ..., w_n, the parser generates $n+1$ sets: s_0, s_1, s_2, ..., s_n, where s_0 is the initial set and s_i is the set for w_i. Each set contains Earley items. An Earley item is written as:

$$[\, A \rightarrow \alpha \cdot \beta,\, i \,]$$

where $A \rightarrow \alpha \cdot \beta$ is a grammar rule in which $\cdot$ denotes the progressive position and i is the reference to the set s_i. The parser operates on s_i in order to build s_{i+1}, until there is no change, according to the following rules:

- **Scanner:** If $[A \rightarrow ... \cdot a \; ..., j]$ is in s_i and $a = w_{i+1}$, then add $[A \rightarrow ... \; a \cdot \; ... \; , j \,]$ to s_{i+1}.
- **Predictor:** If $[A \rightarrow ... \cdot B \; ..., j]$ is in s_i, then add $[B \rightarrow \cdot \, \alpha, i]$ to for all possible α.
- **Completer:** If $[A \rightarrow ... \cdot, j]$ is in s_i, then add $[B \rightarrow ... A \cdot ..., k]$ to s_i for all items $[B \rightarrow ... \cdot A ..., k]$ in s_j.

The parser first initializes s_0 with the top-level grammar rules, then iteratively builds s_1, s_2, ..., s_n. The Earley parsing algorithm is said to be $O(n^3)$ in general case. However, the Earley parsing algorithm is not suitable for parsing CCG because *Predictor* would generate infinite number of $[B \rightarrow \cdot \, \alpha, i]$ as there is, in theory, infinite number of category pairs that can derive *B*. Heuristic search would be necessary if one were to apply the Earley parsing technique to CCG.

2.5 The Present Parser

Currently, there are three well-known CCG parsers: C&C CCG Parser, The OpenCCG Parser, and NLTK CCG Parser. Each parser is explained briefly in the following sections.

2.5.1 C&C CCG Parser

The C & C CCG Parser (http://svn.ask.it.usyd.edu.au/trac/candc/wiki) (Curran, Clark, & Bos, 2007), which is a module of C&C tools, has been developed by James Curran and Stephen Clark. The parser is a wide-coverage parser which can efficiently handle large-scale NLP tasks. The parser is optimized with several techniques. C&C tools also offers the computational semantic tools named Boxer, which has been developed by Johan Bros. Boxer takes a CCG parse as an input and generates a Discourse Representation (Kamp & Reyle, 1993) as an output.

Nevertheless, the parser is not suitable for this project as it is implemented in C++. One of the main purpose in this project is the parser shall be implemented in Python as a module in NLTK. Moreover, the parser is a wide-coverage, integrated with certain optimization technique, and meant to be used for large-scale NLP tasks, while the parser in this project is meant to be an exploration tool and easily extensible.

2.5.2 OpenCCG

The OpenCCG parser (http://openccg.sourceforge.net/) is implemented in Java and meant to be used in the real NLP tasks. The parser applies the multi-modal extensions to CCG (Steedman & Baldridge, 2007; Baldridge & Kruijff, 2003). The multi-modal extension puts modalities on directionality and specifies whether or not it can be applied with the composition or substitution rules. The parser is very practical as it has been used in several other projects.

The OpenCCG parser also offers VisCCG, which is visualization software to create and modify a grammar written in DotCCG format that is the format of CCG Grammar used by the OpenCCG parser.

Nevertheless, the OpenCCG parser is not suitable for this project as it is implemented in Java.

2.5.3 NLTK CCG Parser

The CCG parser in nltk.ccg was first written by Graeme Gange (ggange@csse.unimelb.edu.au). The parser is a CYK chart parser that is neither feature-based nor probabilistic. It merely parses a sentence with categories. The semantic derivation is also not supported. The combinatory rules in (Steedman & Baldridge, 2007), which is also shown in 2.1, are supported.

Gange's parser is integrated with the packed chart technique, though it does not work in an efficient way. This is because it has been decided to pack the edges each of which has:

- Identical category
- Identical span
- Identical combinatory rule which is used to form itself

In fact, if the third condition was dropped, more number of edges would be packed, and the parser would work more efficiently.

The bottom-up chart parsing implemented in the parser also does not work correctly when a type-raising operation is used. Here is the pseudo-code of the bottom-up chart parsing:

```
for span=2 to n
 for start=0 to (n-span)
  for part=1 to span
   left = start
   mid = start + part
   rend = start + span

   for left in { l | l in chart[lstart,mid] }
    if left can be forward type-raised
     add the new edge to chart[lstart,mid]
```

```
 if left can be backward type-raised
  add the new edge to chart[lstart,mid]

for right in { r | r in chart[mid,rend] }
 if right can be forward type-raised
  add the new edge to chart[mid,rend]
 if right can be backward type-raised
  add the new edge to chart[mid,rend]

for left in { l | l in chart[lstart,mid] }
 for right in { r | r in chart[mid,rend] }
  for rule in the combinatory rule (no type-raising)
    if left and right can form a new edge according
       to rule
     add the new edge to chart[lstart,rend]
```

The problem of the pseudo-code above is that the type-raised edges are not processed again in the process. Therefore, some valid parses that contains type-raising might be discarded. For example, if *left* is type-raised and into, say, *new_left*, added to chart, *new_left* will span from *lstart* to *mid* as *left* does. This means that *new_left* will never be processed with the edge spanning from *mid* to *rend* because the set *{ l | l is an edge which spans from lstart to mid }* is already retrieved from chart and *span*, *start*, and *part* will never be the same.

Nevertheless, Gange's parser has implemented the essential structure for CCG, and it is implemented in NLTK. Therefore, the parser implemented in this project extends Gange's parser.

2.6 CCGBank

CCGBank (Hockenmaier & Steedman, 2007) is an important resource for the development of CCG. CCGBank is a corpus, translated from the Penn Treebank, of CCG Derivations. It offers not only syntactic derivations but also word-word dependencies and predicate-argument structures. As CCG derivations are applied on

real-world sentences, many non-standard derivations, which do not comply with any combinatory rules, mentioned in 2.1, have been used. For example,

X X[conj] → X

is one of the several rules designed to handle the coordinate structures in English. Even though CCGBank has used many non-standard rules, while the parser in this project only applies the combinatory rules explained in 2.1, it is still useful to evaluate the correctness of the parser against CCGBank. The evaluation of the parser is explained in Chapter 4.

3 Parser

In this chapter, the parser implemented in this project is explained in detail. The problems are first explained. Then, the purpose of the parser is set. The design and design issues are explained, discussed, and justified. Then, the architecture of the parser and how it fits into NLTK framework is described. The λ-calculus processor is also explained in the architecture section. Later on, the usage of the parser is described. At the end of the section, the unresolved issues are discussed.

3.1 Problem

Even though NLTK already has a CCG Parser, which was implemented by Graeme Gange (ggange@csse.unimelb.edu.au), it is rather a naive one. Besides the minor bugs explained in 2.5.3, there are few drawbacks with the parser.

3.1.1 No Feature and No Feature Unification

The first drawback is that it does not support features and feature unification. Features and feature unification mechanism can largely reduce the complexity of the lexicon. They also enable researchers to explore the richer fragments of CCG. Take subject-verb agreement as an example:

```
Student → NP
Students → NP
Eats → (S\NP)/NP
Eat → (S/NP)/NP
Pig → NP
Pigs → NP
```

According to the lexicon above, 16 sentences are grammatical: *{Student/students} {eat/eats} {pig/pigs}* and *{Pig/pigs} {eat/eats} {student/students}*. If we would like to

eliminate *Student eat {pig/pigs}*, *Students eats {pig/pigs}*, *Pig eat {student/students}*, and *Pigs eats {student/students}*, we had to modify the lexicon to:

```
Student → SNP
Students → PNP
Eats → (S\SNP)/SNP
Eats → (S\SNP)/PNP
Eat → (S\PNP)/PNP
Eat → (S\PNP)/SNP
Pig → SNP
Pigs → PNP
```

The elegance of the lexicon would be lost because *Eats → (S\SNP)/SNP* and *Eats → (S\SNP)/PNP*, and *Eat → (S \PNP)/PNP* and *Eat → (S \PNP)/SNP* seemed to be redundant. On the contrary, with feature and feature unification supported, the feature num, which has either *sg* (singular) or *pl* (plural), can be attached to an *NP*. The subject-verb agreement issue will be solved and the lexicon will not contain any redundancy as shown below:

```
Student → NP[num=sg]
Students → NP[num=pl]
Eats → (S\NP[num=sg])/NP
Eat → (S/NP[num=pl])/NP
Pig → NP[num=sg]
Pigs → NP[num=pl]
```

Please note that *NP[num=sg]* can only unify with either *NP[num=sg]* or *NP* but not *NP[num=pl]*.

A variable feature also plays a crucial role in improving a lexicon's elegance. For example, the article ‘the’ can take either a singular noun phrase or a plural noun phrase. Without variable feature supported, ‘the’ would be defined with two following rules:

```
The → NP[num=sg]/NP[num=sg]
The → NP[num=pl]/NP[num=pl]
```

With a variable feature, the can be defined as:

```
The → NP[num=?x]/NP[num=?x]
```

Please note that *?x* is a variable feature, which can be bound to only one concrete value. With the new rule defined, *The students* will bear the category *NP[num=pl]*, and *The student* will bear the category *NP[num=sg]*.

3.1.2 No Semantic Derivation

The second drawback is that the parser does not support semantic derivation of a sentence. The semantic derivation will benefit many language-understanding tasks as it provides the logical form of a sentence.

3.1.3 No Probabilistic Parsing

The third drawback is that the Gange's parser does not support probabilistic parsing. If a probabilistic parsing is integrated into the parser, it will mainly benefit with these two points: (1) it enables users to explore wider CCG phenomenon, and (2) it provides a pathway and the infrastructure to implement and integrate a new probability model.

3.2 Purpose

In this project, the parser implemented is meant to be used as an experimental tool to explore CCG with NLTK framework. The parser should be extensible by users. The parser should be able to parse a sentence and do semantic derivation given a CCG lexicon, in which a primitive category can have features and a word is associated with its semantic expression. The parser should be able to handle the spurious ambiguity problem better than the previous parser to a certain degree.

3.3 Design

As the parser is meant to be an exploration tool for CCG, there are 3 crucial requirements:

- Users should be able to define their own lexicon, in a convenient way, in which a category can have features and a word is associated with a semantic expression.
- Users should be able to investigate each step of parsing.
- The parser should be easily understandable and extensible.

The lexicon formalism in this project enables users to define primitive categories, families, word-category mapping rules with their features, semantic expressions, and probabilities, and the probabilities of concrete combinatory rules. Families are a short name for a category. They are used for helping users define a lexicon in a more convenient way.

In this project, the CYK parsing is chosen as the parsing algorithm mainly because of its extensibility. Moreover, The Earley parsing is not comfortably applied to CCG as described in 2.4.2. The CYK parsing can be easily extended to support the packed chart technique or the probabilistic parsing technique with beam search. As one of the requirements is that the parser should be easily understandable and extensible, the CYK parsing seems to fit this project more than the Earley parsing does.

The λ-calculus is chosen as the formalism for semantic interpretation because it is widely used for semantic representation. More importantly, it is adopted in (Steedman & Baldridge, 2007).

There are various design issues that arise with CCG. The first issue is which probability model should be used in the probabilistic parsing. There are two famous probability models: the baseline model (Hockenmaier & Steedman, 2002) and word-word dependency model (Collins, 1999). However, those probability models are too complicated for users to define the models manually. Normally, the baseline model and word-word dependency model are induced from a corpus by an algorithm. Therefore, the probability model used in this project is adapted from the probability

model used in a probabilistic context-free grammar. The probability model is explained in 3.3.5.

The second issue is the possibility of forward and backward type-raising operations being applied on any category. A category *X* can be forwardly type-raised to *T/(T\X)* in which *T* can fundamentally be any category. This causes the spurious ambiguity, which, in turn, causes a combinatory explosion of the search space. Therefore, the possibility of forward and backward type-raising operations must be restricted. The restrictions are explained in 3.2.2.

The third issue is to decide whether to implement either atomic feature or indexed feature. Each type of feature has its own advantages and disadvantages. Atomic feature prefers simplicity over flexibility while indexed feature prefers the opposite. However, with atomic feature, more than one constant-valued features are not well-handled. For example, *S[pl,1st]* is rather confusing. Moreover, more than one variable-valued features cannot be coped with atomic feature. One example of multiple variable-valued features is *S[NUM=?x,PER=?y]* in which the variable *?x* and *?y* can be unified with variables or values. As atomic and indexed feature offer benefit in different dimensions, thus both are supported in this project. The mechanism that handles atomic and indexed feature is explained in 3.2.3 and 3.2.4.

3.3.1 Packed Chart

The packed chart (Clark & Curran, 2004) is an optimization technique for a chart parsing. The idea behind the packed chart is rather simple: the edges which have the same category and corresponds to the same part of the sentence are packed into one edge. This prevents the parser from parsing the same structure more than once. Instead of processing all the edges which are structurally the same, the parser processes only once on the packed edge. For a highly ambiguous grammar, such as CCG, the impact of the packed chart technique is very powerful. It has been claimed that the packed chart technique increases the parser's performance by 10 times (Haoliang, Sheng, Muyun, & Tiejun, 2007).

Nevertheless, the packed chart technique introduces a few complications. First, the semantic derivation cannot be processed while the parser parses the sentence because the packed edge does not contain semantic information. It is also impossible for the packed edge to contain semantic information since the edges inside might not share the same semantic information. Instead, the semantic derivation has to be processed during the construction of the parse tree from the chart. The computation of probability is also processed during the construction of the parse tree from the chart with the same reason. Second, it is difficult to retrieve n-best parses except enumerating all the parses first and selecting the n-best ones.

Even though the packed chart technique have caused some complications to the parser, the reduction on time complexity is much more valuable. Therefore, the packed chart technique is also implemented in this project.

In this project, two edges are considered to have the identical structure if they have the identical category whose features are also identical. The implementation of the packed chart technique is in the chart, not the parser. Therefore, the packed chart technique is actually transparent to the parser. When the parser would like to insert a edge into the chart, the chart inserts the edge into its corresponding packed edge. When the parser would like to retrieve an edge from the chart, the chart returns the corresponding packed edge instead. For example, there 5 edges:

```
edge_1 = (S/NP) spanning from 0 to 3

edge_2 = NP spanning from 0 to 3

edge_3 = (S/NP) spanning from 0 to 3

edge_4 = (S/NP) spanning from 0 to 3

edge_5 = S spanning from 0 to 3
```

The packed chart maintains those edges by creating packed edges. For the example above, the packed chart will maintain only 3 edges:

```
p_edge_1 = (S/NP) spanning from 0 to 3

p_edge_2 = NP spanning from 0 to 3
```

```
p_edge_3 = S spanning from 0 to 3
```

Please note that `p_edge_1` has 3 children, which are `edge_1`, `edge_3`, and `edge_4`.

3.3.2 Restriction on Type-Raising

Type-raising is allowed only on primitive categories which are on leaf edges and type-raising on a leaf edge is allowed if its type-raised edge can be combined with one of its consecutive edge with either forward application or backward application. With these restrictions, the type-raising operations will not be explosively used. The restrictions are concluded below:

- **Forward Type-Raising Restriction:** An edge with the category *X* is allowed to be forward type-raised to *T/(T\X)* only if there is an edge with the category *(T\X)* next to its right.
- **Backward Type-Raising Restriction:** An edge with the category *X* is allowed to be backward type-raised to *T\(T/X)* only there is an edge with the category *(T/X)* next to its left.

3.3.3 Atomic and Indexed Feature

Users can use both atomic and indexed feature. The implementation of atomic feature is rather simple. Atomic feature is mapped to indexed feature with '_ATOM_' as its index. In this way, only the mechanism to deal with indexed features is needed.

3.3.4 Feature Unification

Feature unification is the process of unifying features of two categories. Two features can be and will be unified if those features have identical indices with either identical values or at least one of them has variable value. A feature with null value is allowed to be unified with the feature with any value, presuming that they have identical indices. Three examples of successful feature unification are shown below:

- `[NUM=sg] and [NUM=sg]`
- `[] and [NUM=sg] *the first feature set does not contain NUM, therefore the unification is allowed.`
- `[NUM=?x] and [NUM=sg]`

Please note that *?x* is a variable identified with *x*. The example of its use in the forward application rule is shown below:

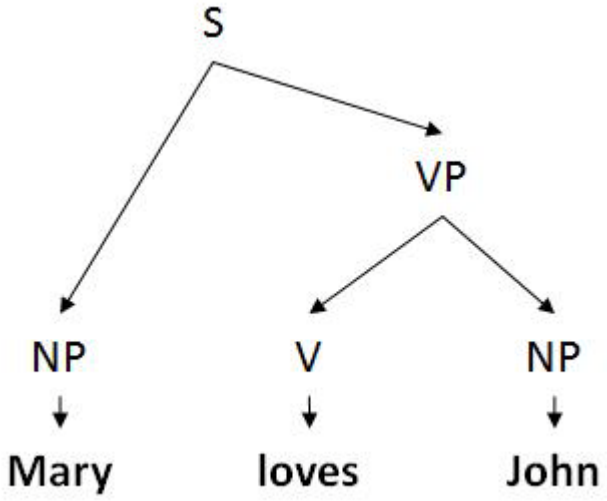
```
S[NUM=?x]/NP[NUM=?x] NP[NUM=sg]  S[NUM=sg]
```

The variable *x* is unified with the constant *sg*. Therefore, the resulting category *S* has the feature *NUM=sg*.

Successful feature unification is a prerequisite for category derivation according to any combinatory rule. With features, the categories, in a combinatory rule, are to be considered identical, only if the categories are identical and their features can be unified.

3.3.5 Probability Model

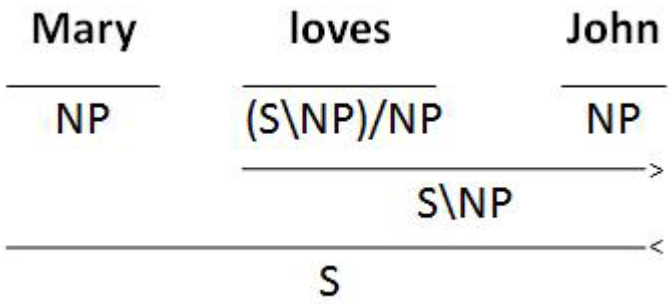
The probability model used in this project is adapted from the probabilistic context-free grammar (PCFG). In PCFG, the probability of a parse is the production of the probabilities of the production rules used. For example,



The tree above has the probability equal to

P(NP → Mary)×P(V → loves)×P(NP → John)×P(VP → V NP)×P(S → NP VP)

The probability of a parse is the production of the probabilities of the word-category mapping rules and the combinatory rules occurred in the parse. For example,



The parse shown above has the probability equal to:

P(Mary → NP)×P(loves → (S\NP)/NP)×P(John → NP)

×P((S\NP)/NP NP → S\NP)×P(NP S\NP → S)

3.3.6 Final Design

The parser shall be implemented as a module in NLTK framework. The parser shall parse a sentence using the CYK parsing algorithm and give n-best parses, in accordance to their probabilities as a result, where n can be adjusted by users. A parse shall contain semantic derivation at each step of the derivation process. The λ-calculus is used as the formalism for semantic derivation. The parser shall apply the packed chart technique and put the restrictions, explained in 3.2.2, on the type-raising operations in order to prevent a combinatory explosion of the search space. Both atomic and indexed features are supported for convenient use.

The lexicon shall be defined by users, but the combinatory rules are hard-coded and cannot be defined by users. The lexicon formalism supports defining primitive categories, families and word-category mapping rules in which a primitive category can have features and a word is associated with a semantic expression.

3.4 Implementation

The parser is implemented based on the previous CCG parser in *nltk.ccg* implemented by Graeme Gange (ggange@csse.unimelb.edu.au). Gange's parser architecture is explained in 2.5.3. The λ-calculus processor is also already implemented in *nltk.sem.logic*.

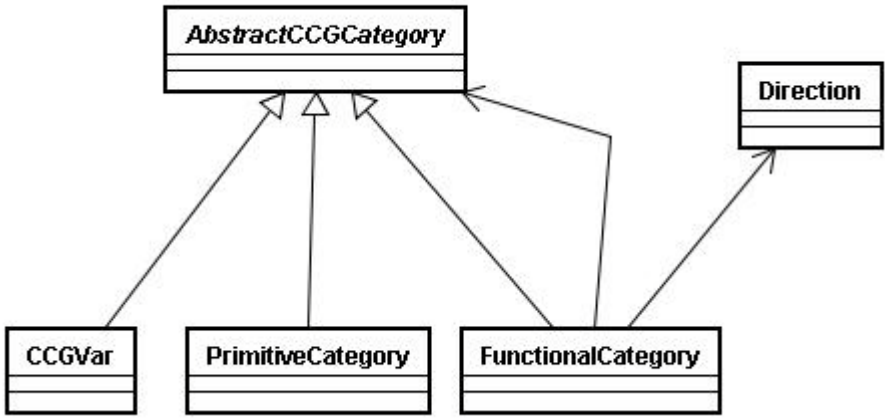
3.4.1 Parser Implementation

This subsection explains the implementation of categories, the combinatory rules, the lexicon, the chart, and the parser.

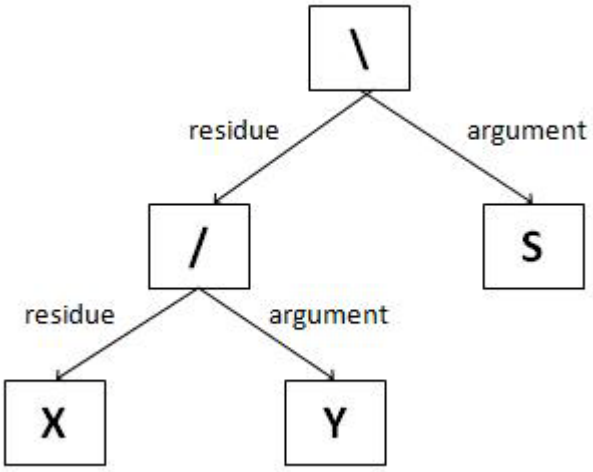
3.4.1.1 Category Implementation

Categories are implemented with the composite pattern. The classes involved are *AbstractCCGCategory*, *PrimitiveCategory*, *FunctionalCategory*, *Direction*, and *CCGVar*. *PrimitiveCategory* and *FunctionalCategory* represent primitive categories and complex categories, respectively. *CCGVar* represents generic categories which

can be unified with an instance of *PrimitiveCategory* or *FunctionalCategory*. *CCGVar* is used with the 'and' example described above. *FunctionalCategory* holds two instances of *AbstractCCGCategory*; the first instance is the residue once it is applied with the argument, which is the second instance. *FunctionalCategory* also holds an instance of *Direction* to indicate whether it is a left function or a right function. The class diagram of these classes is shown below:



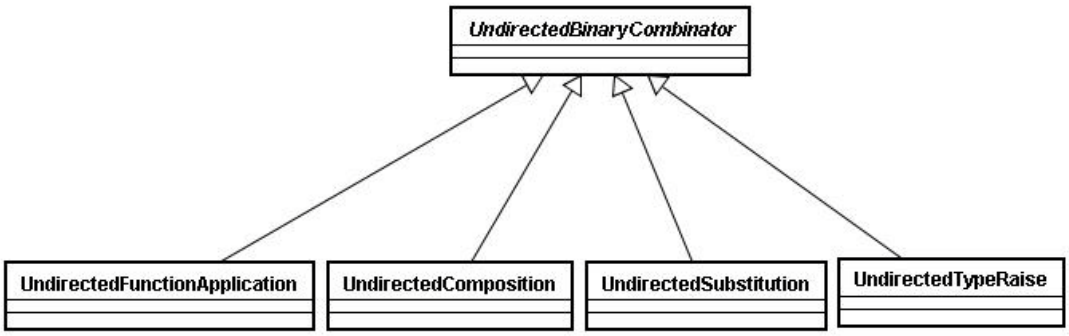
For example, the category *(X/Y)\Z* has the object hierarchy as shown below:

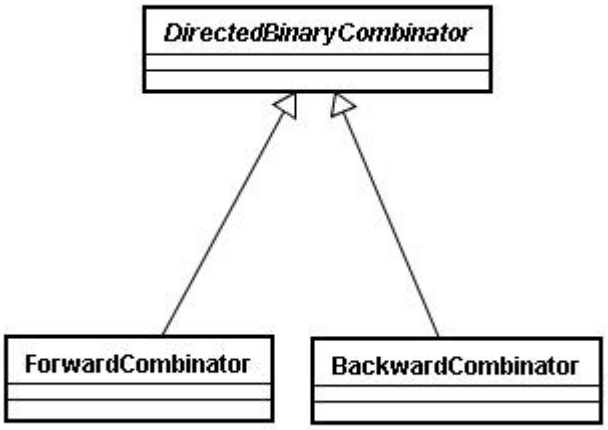


The topmost object is an instance of *FunctionalCategory* which has direction as backward. It holds residue, which is another instance of *FunctionalCategory*, and argument, which is *S*. The instance of *FunctionalCategory* on the second level holds residue, which is *X*, and argument, which is *Y*.

3.4.1.2 Combinatory Rule Implementation

A combinatory rule is implemented with *DirectedBinaryCombinator*, which is the superclass of *ForwardCombinator* and *BackwardCombinator*. They define a direction of the rule. *DirectedBinaryCombinator* holds a constraint predicate and an instance of *UndirectedBinaryCombinator*, which is the superclass of *UndirectedFunctionApplication*, *UndirectedComposition*, *UndirectedSubstitution*, and *UndirectedTypeRaise*. They define an operation of the rule. The class diagram is shown below:





For example, the backward composition rule is an instance of *BackwardCombinator*, which holds an instance of *UndirectedComposition*.

3.4.1.3 Parser Architecture

The chart is represented with the class *CCGChart*, which subclasses *Chart* from *nltk.parse*. The chart supports operations, used by *CCGChartParser*: *select(i,j)* and *insert(new_edge, left_child,right_child)*. *select(i,j)* return all edges that span from *i*-th word to *j*-th word. *insert(new_edge, left_child,right_child)* adds *new_edge* to the chart and associates *new_edge* with *left_child* and *right_child*.

The parser is implemented with the class *CCGChartParser*. The parser works in a bottom-up manner. It first creates leaf nodes from each word. Then, it exhaustively creates nodes that can be a parent of a pair of consecutive nodes according to the combinatory rules. Please note that a node holds only a category. Therefore, the computation complexity of the parsing operation is $O(rn^5)$ where n is the number of words and r is the number of combinatory rules.

3.4.1.4 Lexicon Implementation

The lexicon is implemented with the class *CCGLexicon* in *nltk.tnn_ccg.lexicon*. *CCGLexicon* simply keeps the list of primitive categories, the list of families, the list of word-category mapping rules, and the list of concrete combinatory rules with their probabilities. The important function to be described here is *parseLexicon()* that turns a string into an instance of *CCGLexicon*. The process of parsing a lexicon mainly uses regular expressions which are defined in the top of *nltk.tnn_ccg.lexicon*. Please take a look into *nltk.tnn_ccg.lexicon* in order to gain understanding how a lexicon is parsed.

The list of primitive categories is implemented with a python list in which each primitive category is a simple string. The list of families is implemented with a python dictionary in which a family name is mapped to an instance of *AbstractCCGCategory*. The implementation of categories is explained later in this section. The list of word-category mapping rules is also implemented with a python dictionary in which a word is mapped to an instance of *LexiconWordEntry*, which holds a category, a semantic predicate (in the form of simple string), and a probability.

3.4.1.5 Feature Unification Implementation

The feature unification occurs on the primitive-category level since only primitive categories have features. Please note that a functional category does not have feature.

The two relevant methods in the *PrimitiveCategory* class are *can_unify(other)* and *substitute(subs)*. *can_unify(other)* indicates whether or not the class instance can be unified with other. If it can be unified, then *can_unify(other)* returns two lists: One for category variables with their unified category, another is for feature variables with their unified value. *substitute(subs)* created a copy of the class instance with its variables unified to the value specified in *subs*, which are the lists returned from *can_unify(other)*. Please note that *substitute(subs)* had an effect only on the *CCGVar* class because *PrimitiveCategory* and *FunctionalCategory* are static.

The function *unify_feature(a,b)* has been added in order to unify two feature sets *a* and *b*. If *a* and *b* can be unified, the list of feature variables and their unified values is returned, otherwise the function returns null.

For full source code, please see *nltk.tnn_ccg.api*.

3.4.1.6 The Packed Chart Technique Implementation

The packed chart technique is implemented in *Chart*, which is in *nltk.parse.chart*. When an edge is inserted into the chart, the chart checks whether or not the edge is considered as equal to some other edges, which is already in the chart. If so, the new edge is packed with those equal edges. If not, the new group is formed.

Therefore, to implement the packed chart technique, the two classes are devised: *CCGEdge* and *CCGLeafEdge*. In this project, two edges are considered to have the identical structure if they have the identical category whose features are also identical and they span in the same range. Their *__cmp__(self, other)* and *__hash__(self)* are overridden to reflect the mentioned condition.

For full source code, please see *PackedCCGChart* in *nltk.tnn_ccg.pchart*.

3.4.1.7 Semantic Derivation Implementation

The semantic derivation is being done during the retrieval of the valid parses from the chart. When an edge is formed, its semantic expression is computed based on its children. The function which does semantic derivation is *computeSem(children,parent,rule)*. Since a packed edge does not contain information about the combinatory rule used to form an edge, the combinatory rule used to form an edge can be computed using *determineRule(lhs,left,right)*.

For full source code, please see *nltk.tnn_ccg.pchart*.

3.4.2 The λ-calculus Processor Implementation

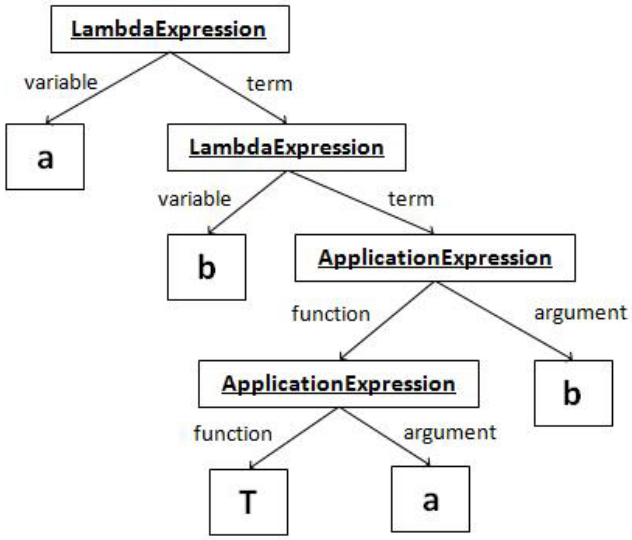
The λ-calculus processor is implemented in *nltk.tnn_sem.logic*. It supports parsing a λ-calculus expression, turning a string into a managed structure in Python. It also supports application of λ-calculus function with constant value. However, it does not support composition between two λ-calculus functions.

In *nltk.tnn_sem.logic*, *LogicParser* is the class which parses a string into an instance of *Expression*. The subclasses of *Expression* represent different types of predicates: *LambdaExpression* represents a λ-calculus expression, *ApplicationExpression* represents a functional expression applying an argument expression, *IndividualVariableExpression* represents a value-type variable, and *FunctionalVariableExpression* represents a functional-type variable. Please note that in this project we focus only on the λ-calculus processor. There are classes representing AND-expression, OR-expression and etc., which are not described here.

LambdaExpression recursively organizes a λ-calculus expression. *LambdaExpression* holds *variable* and *term*; *variable* can be of *IndividualVariableExpression* or *FunctionalVariableExpression*. *term* can be of any subclass of *Expression*.

ApplicationExpression also recursively organizes a functional expression and an argument expression. *ApplicationExpression* holds *function*, which represents a functional expression, and *argument*, which represents an argument expression.

For example, *λa λb . T(a,b)* is organized in the structure shown below:



The λ-calculus processor supports the application operation. The application operation can be used by parsing a predicate with *LogicParser.parse()* and invoking *simplify()* in order to obtain the simplest equivalent form. The example is shown below:

```
>> str = "\x.man(x) (john)"
>> x = LogicParser().parse(str).simplify()
>> print x
man(john)
```

Moreover, it also supports the application operation where a functional-type variable is being bound to the argument. The example is shown below:

```
>> str = "\T x. T (x) (\x . test(x))"
>> x = LogicParser().parse(str).simplify()
>> print x
\x . test(x)
```

3.5 Usage

To use the parser, the lexicon and a set of combinatory rules must be first defined. The instruction how to define a lexicon is explained in 3.4.1. The instruction how to use the parser is rather simple: (1) instantiate the class *CCGChartParser* in *nltk.tnn_ccg.pchart* with a lexicon and a set of combinatory rules (optional), and (2) use the *method nbest_parse(...)* to parse a sentence. *nbest_parse(...)* returns a list of valid parses. A parse is in a form of tree structure, which can be printed with the function *printCCGDerivation(...)*. You can find the method's signature in *nltk.tnn_ccg.pchart.CCGChartParser*.

The feature worth noting is that *nbest_parse(...)* can be configured with 1 option:

- **Number of Parses:** to limit the number of returned parses; 0 means output all valid parses.

To limit the number of parses returned, the parameter n is used; *n=100* means that the number of parses returned will not exceed 100 parses. The default option is *n=0,* which means the parser shall return all valid parses.

The example of using the parser is shown below:

```
>> lex = parseLexicon(r'''
     :- S,NP
     He => NP {sem=\x.he(x)} [1.0]
     Walks => S\NP {sem=\X. walk(X)} [1.0]
     There => S\S {sem=\x . there(x)} [1.0]
   ''')
>> parser = CCGChartParser(lex)
>> all_parses = parser.nbest_parse(“He walks
there”.split(),n=100)
>> for parse in all_parses:
      printCCGDerivation(parse)
```

3.5.1 CCG lexicon formalism

The CCG lexicon formalism supports defining primitive categories, families, and word-category mapping rules in which a primitive category can have features and a word is associated with a semantic expression.

First of all, the set of primitive categories must be defined. The line to define a set of primitive categories shall be started with :-, followed by primitive categories separated by ',' (comma). The first category defined is the target category; if a sentence is derived into the target category, the sentence is considered valid according to the defined lexicon. The example of defining a set of primitive categories with S as the target category is shown below:

```
:- S, NP, PP, VP
```

After a set of primitive categories is defined, the families shall be defined. Families are used as shortcuts to facilitate the process of defining a lexicon. The format for defining a family is shown below:

```
<family_name> => <category> {sem=<semantic_exp>}
```

After the families are defined, the word-category mapping rules can be defined. The format of a word-category mapping rule is shown below:

```
<word> => <category> {sem=<semantic_exp>} [<prob>]
```

In case of a word is mapped to a family, the format is:

```
<word> => <family_name> {sym=<symbol>}[<prob>]
```

<*symbol*> will replace the macro *SYM* defined in the family <family_name>. Please note that a semantic predicate must be written to be compatible with its corresponding category as explained in 2.3 and <*prob*> is the probability of the word being mapped to the category.

The example of family being used is shown as follow. If the family *TransitiveVerb* is defined as follow:

```
TransitiveVerb :: (S\NP)/NP {sem=\a b . SYM(b,a)}
```

Moreover, the verbs which have the category *(S\NP)/NP* can be conveniently defined with *TransitiveVerb*. The example is shown below:

```
Eat => TransitiveVerb {sym=eat} [1.0]
Walk => TransitiveVerb {sym=walk}[1.0]
Learn => TransitiveVerb {sym=learn}[1.0]
```

The three examples above are equivalent to the lengthy version shown below:

```
Eat => (S\NP)/NP {sem=\a b . eat(b,a)} [1.0]
Walk => (S\NP)/NP {sem=\a b . walk(b,a)} [1.0]
Learn => (S\NP)/NP {sem=\a b . learn(b,a)} [1.0]
```

Another point to mention is that *<semantic_exp>* can contain *SYM*. *SYM* is a macro word which will be substituted later with a value defined in a word-category mapping rule. For example, the family *TransitiveVerb :: (S\NP)/NP {sem=\a b . SYM(b,a)}* is used in *Eat => TransitiveVerb {sym=eat}*, therefore, *SYM* is substituted with eat as defined with *{sym=eat}*.

The modality of a slash can be defined with ',' (comma) and '.' (dot). Comma is for preventing composition and dot is for preventing substitution. Their use is shown below:

- To define $/_{*}$, which forbids order-preserving associativity and limited permutation, both comma and dot are used, for example, /.,.
- To define $/_{\diamond}$, which permits only order-preserving associativity, dot is used, for example, /..
- To define $/_{\times}$, which allows only limited permutation , comma is used, for example, /,.
- To define /., which allows any combinatory rule to be applied, a plain slash, for example, /, is used

As the parser is a probabilistic one, the probability of a concrete combinatory rule must be defined. The format to define the probability of a concrete combinatory rule is:

```
<rule> [<prob>]
```

The examples are shown below:

```
(NP/N) (NP\(NP/N)) -> NP [0.2]
(NP/N) N -> NP [0.8]
(S/(S\NP)) (S\NP) -> S [0.2
NP (S\NP) -> S [0.8]
```

The example of a complete lexicon is shown below:

```
:- S,NP,N

IntransVsg :: S\NP[NUM=sg] {sem=\x.SYM(x)}
IntransVpl :: S\NP[NUM=pl] {sem=\x.SYM(x)}
TransVsg :: S\NP[NUM=sg]/NP {sem=\x y.SYM(y,x)}
TransVpl :: S\NP[NUM=pl]/NP {sem=\x y.SYM(y,x)}

John => NP {sem=john}
Mary => NP {sem=mary}
sleeps => IntransVsg {sym=sleep}
gives => (S\NP)/NP/NP {sem=\a b c.give(from(c),b,to(a))}
on => ((S\.,S)/NP) {sem=\x y.on(y,x)}
the => NP/,.N {sem=\x.the(x)}
bed => N {sem=bed}

((S\NP)/N) N -> (S\NP) [0.45]
((S\NP)/NP) ((S\NP)\((S\NP)/NP)) -> (S\NP) [0.10]
((S\NP)/NP) NP -> (S\NP) [0.45]
(NP/N) (NP\(NP/N)) -> NP [0.2]
(NP/N) N -> NP [0.8]
(S/(S\NP)) (S\NP) -> S [0.2]
NP (S\NP) -> S [0.8]
```

3.6 Unresolved Issues

There are two issues that have not been resolved with the implementation of the parser in this project. The first immediately noticeable issue is the variable category issue. Variable categories are not supported by this parser but supported by the Gange's parser. The variable categories feature has been taken out because of its

semantic compatibility problem. The second issue is the problem of defining a lexicon.

3.6.1 Variable Category Issue

Variable categories are not officially supported anymore because of the semantic compatibility problem. It is not possible to write a semantic predicate compatible with a variable category or a complex category that contains variables. For example, the category

```
var/var
```

has a semantic predicate in the form of

```
λx . F(x) where F is a function
```

if *var* is unified with a primitive category. If *var* is unified with a complex category, for example, *(S/NP)*, the category will be:

```
(S/NP)/(S/NP)
```

whose semantic predicate must be in the form of:

```
λT x. F(T,x) where F is a function
```

Nevertheless, variable categories are convenient for defining conjunctions, even though they are not fully compatible with the parser. If users would like to ignore semantic derivation, variable categories can be used with empty semantic predicate. For example,

```
and => (var\var)/var {sem=}
```

Please be aware that if the semantic predicate is not empty, the parser might fail.

3.6.2 Lexicon Issue

In this project, a probabilistic parsing is not integrated into the parser due to the problem of defining a lexicon. As a probabilistic parsing is integrated, every possible

concrete combinatory rule needs to be defined. For example, a lexicon with three words defined:

```
he → NP
eats → (S\NP)/NP
meat → NP
```

must define the probabilities for *S → NP S\NP, (S\NP) → (S\NP)/NP NP, NP → he, (S\NP)/NP → eats, and NP → meat* because these rules are used in the derivation. With a lexicon with many words, it is very difficult to define the probability for each possible rule manually.

With the parser in this project, the lexicon is still needed to define manually. Nevertheless, the lexicon can report which rules are missing to help users identify which rules are missing. Please take a look into *nltk.tnn_ccg.pchart* for how to get the report.

For a lexicon with many words, users are encouraged to use an algorithm to generate the lexicon instead.

4 Parser Evaluation

The evaluation of a parser usually concerns three main issues: correctness, coverage and efficiency. The correctness of the parser can be directly measured by whether or not the parser produces all the valid parses for a sentence, given a lexicon. The coverage of the parser with regard to a corpus can be measured by counting how many sentences in the corpus are accepted by the parser. The efficiency of the parser concerns the space and time used by the parser.

To evaluate the parser against other parser is also an option. However, other CCG parsers, for example, C & C CCG parser and OpenCCG, are a wide-coverage parser, while the parser in this project is not. Therefore, it is not sensible to evaluate against those parsers with respect to a corpus.

Moreover, to evaluate the coverage of the parser for CCGBank does not make much sense. To parse a sentence in CCGBank using the lexicon provided by CCGBank requires a parser to allow many non-standard combinatory rules. Some of the non-standard combinatory rules are listed below:

```
N → NP
S\NP → (S\NP)\(S\NP)
NP → S[dcl]
NP[nb]/N → NP
S/S → S[dcl]
(S/S)/NP → S/S
N/N → N
S[adj]\NP → NP\NP
X X[conj] → X, where X can be any category
, X → X[conj], where X can be any category
```

Some of the rules used in CCGBank allow a category to combine with a symbol to form another category. Therefore, as the parser allows only standard combinatory rules shown in 2.1, the coverage of the parser for CCGBank is guaranteed to be very low.

Nevertheless, in this project, the correctness of the parser is evaluated by feeding the parser with grammatical sentences and see whether or not the parser accepts those sentences. The efficiency of the parser is also evaluated by counting the number of steps taken to parse sentences.

The unit test has also been done to ensure the correctness of the parser's algorithm, but the description of the unit test is not included in this report.

4.1 Correctness Evaluation

As this is a CCG parser, it is very difficult to evaluate the correctness of the parser directly. This is because of the spurious ambiguity problem, which often arises in CCG. It is infeasible to define all the valid parses for a sentence manually. Moreover, there is, at the moment, no corpus which provides all the valid parses for a sentence, and it is very unlikely that there will be one. Therefore, to evaluate whether or not the parser produces all the valid parses for a sentence is infeasible.

As CCGBank provides one derivation for each sentence, one alternative is to evaluate whether or not the parser produces the same derivation. However, this approach cannot evaluate whether or not the parser produces false-positive parses. Nevertheless, the correctness of the parser can still be measured, to certain degree, by the mentioned approach. Therefore, the approach is used for evaluating the correctness of the parser.

The goal of this evaluation is to count how many sentences are accepted, how many are rejected, and how many from those accepted sentences the parser correctly assigns the categories.

Evaluation Setting

The selected 152 sentences from *wsj_0001.auto*, *wsj_0002.auto*, ..., and *wsj_0099.auto* in CCGBank are chosen based on the below conditions:

- The sentence does not contain any word corresponding to the category *X[conj]*, where *X* can be any category.
- The sentence corresponds to *S[dcl]*
- The sentence does not contain these symbols: semi-colon, colon, comma, or question-mark

Please note that there are in total 1,913 sentences and, with the above conditions, 1,760 sentences are discarded and 153 sentences are selected.

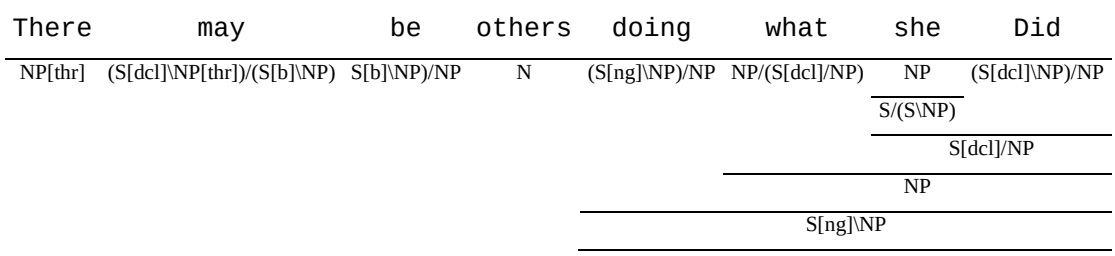
These conditions are set because, in CCGBank, a derivation might contain non-standard combinatory rules, for example,

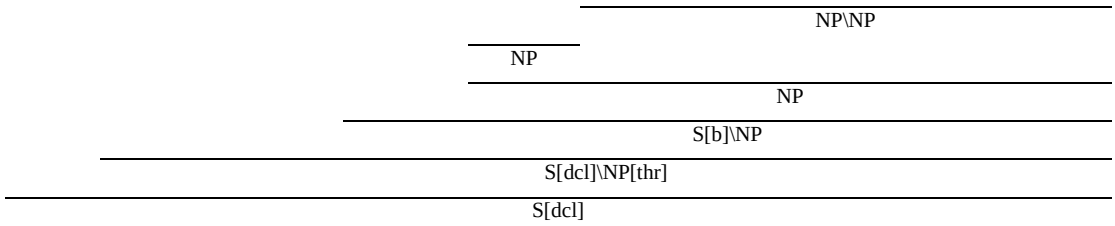
```
NP NP[conj] → NP
```

Even some sentences are discarded; the parser still needs some modification. The parser is modified to allow some non-standard combinatory rules as shown below:

```
N → NP
S\NP → (S\NP)\(S\NP)
NP → S[dcl]
NP[nb]/N → NP
S/S → S[dcl]
(S/S)/NP → S/S
N/N → N
S\NP → NP\NP
```

The above non-standard rules are required to be allowed because most of the sentences in CCGBank use them; otherwise the parser would fail on almost every sentence and this evaluation would not prove the correctness. Moreover, the period at the end of each sentence is also removed. An example of the derivation which uses non-standard rules is shown below:





There are three non-standard combinatory rules have been used in the example above, which are:

```
N → NP
S[ng]\NP → NP\NP
S/(S\NP) (S[dcl]\NP)/NP → S[dcl]/NP
```

The parser uses the lexicon defined in *lexicon.wsj00.txt* in CCGBank. The rules which do not map a word to a category are discarded, for example,

```
! → .
, → ,
- → :
; → ;
```

The rules shown above are the example of the rules that are discarded. This is because the parser does not support special category like those.

All the sentences used in this evaluation are in nltk.tnn_ccg.tested_sentences.txt.

Evaluation Result and Analysis

134 sentences are accepted, while 19 sentences are rejected. 18 out of the 19 rejected sentences are due the fact that the lexicon provided by CCGBank itself does not have a corresponding category for numbers in the sentences. For example, for the sentence,

```
The $ 35.7 million net loss equals 86 cents a share
```

The lexicon does not have a corresponding category for the number *86*. Other rejected sentences also contain numbers which is not contained in the lexicon.

The other one is rejected because the double-quote in it is mapped to the special category *LRB*, which cannot be handled by this parser.

In each sentence from all the 134 accepted sentences, there is a category assignment pattern, produced by the parser, corresponding to the one provided in CCGBank.

The result of this evaluation can be seen in nltk.tnn_ccg.tested_sentences.txt.

This has proved that the parser correctly accepts grammatical sentences and correctly assigns the category. However, this evaluation does NOT prove that the parser:

- × Correctly rejects ungrammatical sentences
- × Correctly produces all the valid parses
- × Does not produce invalid parses

4.2 Performance Evaluation

In order to evaluate the performance of the parser, instead of measuring the elapsed time to parse each sentence, the number of the steps taken in the chart parsing algorithm is measured. Since the parser is written in Python, it is run on a virtual machine with automated memory management. It would be inaccurate to measure the elapsed time because the automated memory management could disrupt the measurement. Other running processes on the same machine could also disrupt the measurement. Therefore, to count the number of the steps is far more accurate.

The goal of this evaluation is to count the steps of the chart parsing algorithm and analyzed it against the length of the sentence. Gange's parser is also evaluated as it has been criticized about its inefficient-enough packed chart technique in 2.5.3.

Evaluation Setting

A toy lexicon and a sequence of sentences are made. They are built in the way that will cause the spurious ambiguity problem. The lexicon is shown below:

```
:- S
x => S/S {sem=} [1.0]
```

```
y => S\S {sem=} [1.0]
z => S {sem=} [1.0]
```

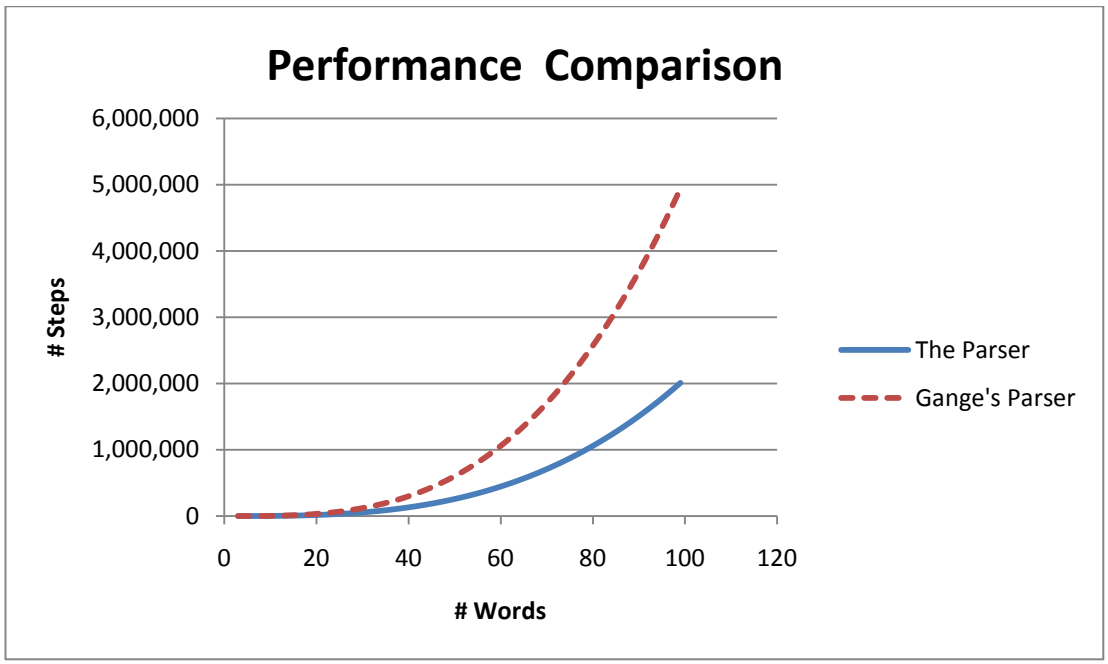
The sequence of 100 sentences is shown below:

```
x z
x y z
x y x z
x y x y z
x y x y x z
x y x y x y z
.
.
.
```

The two parsers are fed with the lexicon above and the sentences. The type-raising operations are not allowed because Gange's parser does not implement it correctly as explained in 2.5.3.

Evaluation Result and Analysis

The number of steps is compared against the length of a sentence as shown below:



The number of steps grows exponentially compared to the length of sentence. This is reasonable because the number of generated edges in the chart also grows exponentially compared to the length of sentence.

As explained in 2.5.3, Gange's parser has inefficiently implemented the packed chart technique. Gange's parser does not pack the edges which have identical categories but result from different combinatory rule, while the parser in this project does. The difference in performance between the parser in this project and Gange's parser is obvious from the given graph. The new parser clearly outperforms Gange's parser.

5 Conclusion

In this project, the CCG parser (*nltk.tnn_ccg*) in NLTK has been further developed based on Gange's parser (*nltk.ccg*), which has 3 main drawbacks.

The first drawback is that Gange's parser does not support feature and feature unification. Allowing a category to have features will enable users to explore richer fragments of CCG. It also reduces the complexity of the lexicon as explained in 3.1.2.

The second drawback is that the Gange's parser does not support semantic derivation, which is essential to other NLP-related tasks as it provides the logical form of a sentence.

The third drawback is that Gange's parser does not support probabilistic parsing. A probabilistic parsing provides the wider possibility to experiment with CCG's phenomenon.

Besides the drawbacks, Gange's parser has also implemented the packed chart technique in an inefficient way. Moreover, the type-raised edges are sometimes ignored by Gange's parser as explained in 2.5.3.

The new parser implemented in nltk.tnn_ccg has been improved based on the drawbacks and the mistakes explained above. The parser also supports several new features: The parser supports both atomic and indexed feature. It supports semantic derivation. It is integrated with the packed chart technique in order to handle the spurious ambiguity problem efficiently. It also supports probabilistic parsing. The probability model adopted by the parser is adapted from the standard probability model used by a probabilistic context-free grammar.

The parser still has two unresolved issues. The first unresolved issue is that it is not possible to write a semantic expression which is compatible with a category that contains variable(s). This is mainly because a variable might be unified to any form

of complex category. Therefore, it is impossible to write a semantic expression which is compatible with every possible form of complex category.

The second unresolved issue is the difficulty of building a lexicon. As a probabilistic parsing is integrated, every possible concrete combinatory rule needs to be defined. For example, a lexicon with three words defined:

```
he → NP
eats → (S\NP)/NP
meat → NP
```

Users must, at least, define the probabilities for *P(NP S\NP → S)*, *P((S\NP)/NP NP → S\NP)*, *P(he → NP)*, *P(eats → (S\NP)/NP)*, and *P(meat → NP)* because these rules are used in the derivation. With a lexicon with many words, it is very difficult to define the probability for each possible rule manually.

The parser has been evaluated for its correctness as described in Chapter 4. It is evident that the parser does work correctly. However, the parser is not ready to be used in real world. To parse a real-world sentence, the parser needs to cope with non-standard CCG rules in order to handle coordination, extraposition of appositives, and other linguistic phenomenon, which are described in (Hockenmaier, 2003).

The parser has also been evaluated for its performance. The packed chart technique does have an effect on the performance of the parser as the number of steps grows slower while the number of words increases. The parser is also compared with Gange's parser. It clearly outperforms Gange's parser, especially when the spurious ambiguity problem arises.

5.1 Future work

The parser can be extended to support real-world sentences from CCGBank by allowing non-standard combinatory rules to be defined in the lexicon. This will make the parser more practical as it can be used within real-world setting. To improve even more on practicality, the parser should have a built-in ready-to-use lexicon, though

the lexicon is language-dependent. This will enable others to use the parser as an external component for their systems.

Alternatively, to allow all combinatory rules to be defined by users will definitely benefit researchers. This will surely increase the possibility for researchers to explore higher levels of CCG phenomenon.

Another possible, though doubtful, improvement is to implement the inside chart parsing algorithm in order to improve the parser efficiency, especially, when the parser needs to return only n-best parses, because the inside chart parsing can return the parses immediately after n-best valid parses have been found. However, there are two main drawbacks; the inside chart parsing is not efficient than the normal chart parsing when n specified is equal or greater than the number of valid parses of a sentence. Another drawback is that it is very difficult to integrate the inside chart parsing with the packed chart technique. Therefore, it is not obvious that the inside chart parsing has advantages over the normal chart parsing.

Bibliography

Baldridge, J., & Kruijff, G.-J. M. (2003). Multi-Model Combinatory Categorial Grammar. *10th Conference of the European Chapter of the Association for Computational Linguistics.*

Bird, S., Klein, E., & Loper, E. (2009). *Natural Language Processing with Python --- Analyzing Text with the Natural Language Toolkit.* O'Reilly Media.

Carroll, J., Briscoe, E., & Sanfilippo, A. (1998). Parser evaluation: a survey and a new proposal. *Proceedings of the 1st International Conference on Language Resources and Evaluation*, (pp. 447-454).

Clark, S., & Curran, J. R. (2004). Parsing the WSJ using CCG and log-linear models. *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics.*

Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing.* PhD Thesis.

Curran, J. R., Clark, S., & Bos, J. (2007). Linguistically Motivated Large-Scale NLP with C&C and Boxer. *Proceedings of the ACL 2007 Demonstrations Session*, (pp. 29-32).

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM* , 94-102.

Haoliang, Q., Sheng, L., Muyun, Y., & Tiejun, Z. (2007). Packed Forest Chart Parser. *Journal of Computer Science , 3* (1), 9-13.

Hockenmaier, J. (2003). *Data and Models for Statistical Parsing with Combinatory Categorial Grammar.* PhD Thesis.

Hockenmaier, J., & Steedman, M. (2007). CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics , 33* (3), 355-396.

Hockenmaier, J., & Steedman, M. (2002). Generative Models for Statistical Parsing with Combinatory Categorial Grammar. *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics.*

Kamp, H., & Reyle, U. (1993). *From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Springer.

Kasami, T. (1965). *An Efficient Recognition and Syntax-analysis Algorithm for Context-Free Languages.* Hawaii: Hawaii University.

Steedman, M. (2002). *Plans, Affordances, and Combinatory Grammar.* Retrieved from ftp://ftp.cogsci.ed.ac.uk/pub/steedman/affordances/pelletier.pdf

Steedman, M., & Baldridge, J. (2007). *Combinatory Categorial Grammar.* Retrieved from ftp://ftp.cogsci.ed.ac.uk/pub/steedman/ccg/manifesto.pdf

Younger, D. H. (1967). Reconition and Parsing of Context-free Languages in time O(n^3). *Information and Control* , 189-208.