433-460 Project: Chart Parsing for Combinatory Categorial Grammars

Graeme Gange

June 3, 2008

1 Introduction

Much of the literature on parsing languages focuses on context-free grammars (CFGs). CFGs have a number of useful characteristics – they are conceptually simple, and have reasonably efficient parsing algorithms. They are, however, limited to the context-free languages – there are a wide variety of linguistic phenomena that CFGs are simply unable to represent. Also, an ideal grammar formalism would have derivations that provided insight into the processes underlying the language.

Combinatory categorial grammars[4] are a formalism developed address these issues. Rather than explicitly building derivation rules from a specified grammar, CCG derivations use a fixed set of rules (so-called combinators). Each word is associated with a category which represents the syntactic role of the word; each derivation step is the result of applying a combinator to one or more (generally two) adjacent categories to construct a new category. CCGs have been demonstrated to be capable of expressing a variety of linguistic phenomena.[2]

A CCG consists of two components – a mapping between each word and a set of possible categories (the lexicon), and the set of permitted combinators. Each category is either a primitive category, such as S, NP or VP, or a function which takes an argument to produce a category; these functional categories are marked with a slash to denote the direction of application.

For example, the category of determiners is NP/N: given a noun as an argument to the right, it becomes a noun-phrase. Intransitive verbs, however, have the category $S\backslash NP$: given a noun-phrase to the left, it may form a sentence.

Any grammar formalism, however expressive, is worthless without an efficient method for parsing. While the structure of CCGs is somewhat prohibitive to top-down parsing, it proves well-suited to bottom-up parsing. The purpose of this project is to construct a bottom-up chart parser for CCGs.

2 Method

2.1 Combinators

The defining characteristic of the combinatory categorial grammar is, of course the combinator. CCGs use function application together with a set of 3 other combinators – composition (B), substitution (S) and type-raising (T).

$$X/Y\ Y \Rightarrow X\ (>)$$
$$Y\ X\backslash Y \Rightarrow X\ (<) \qquad (1)$$

Figure 1: Derivation rules for forward and backward function application.

$$X/Y\ Y/Z \Rightarrow X/Z\ (>B)$$
$$Y\backslash Z\ X\backslash Y \Rightarrow X\backslash Z\ (<B) \qquad (2)$$
$$Y/Z\ X\backslash Y \Rightarrow X/Z\ (<B\times)$$

Figure 2: Derivation rules for variations of function composition.

2.1.1 Function Application

Function application is the most basic operation used in CCG derivations. Given a functor and an argument, it returns the result of applying the argument to the functor. The derivation rules for this are given in Figure 2.1.1.

2.1.2 Composition

The composition (B) combinator is conceptually the same as function composition in functional programming. Two functions are combined by redirecting the output from the first to the input for the second. That is, $(f.g)\,x \Leftrightarrow f(g\,x)$.

The derivation rules for variations of composition used in English are given in Figure 2.1.2. An important point is that the $< B\times$ combinator is restricted such that the variable Z must be a primitive category. Note that there is also a forwards crossing B combinator, but it is not used in English derivations.

Note that the current implementation does not include the generalised B combinators, such as B^2, used for composing into ditransitive verbs, for example '$\text{will}_{(S\backslash NP)/VP}$ $\text{give}_{(VP/NP)/NP}$':

$$(S\backslash NP)/VP\ \ (VP/NP)/NP \Rightarrow ((S\backslash NP)/NP)/NP \qquad (3)$$

2.1.3 Substitution

Substitution is a combinator necessary for modelling multiple dependencies, such as 'file without reading'. The derivation rules for substitution are given in Figure 2.1.3.

As with the $< B\times$ combinator, both the $> S$ and $< S\times$ combinators are restricted such that the category Z must be primitive. As with the B combinators, there are also $< S$ and $> S\times$ variations, but they do not occur in English derivations.

2.1.4 Type Raising

Type raising is needed in many cases to allow coordination of categories which would otherwise be impossible. An example of this is 'I_{NP} $\text{will}_{(S\backslash NP)/VP}$',

$$(X/Y)/Z\ Y/Z \Rightarrow X/Z\ (>S)$$
$$Y/Z\ (X\backslash Y)/Z \Rightarrow X/Z\ (<S\times) \quad (4)$$

Figure 3: Derivation rules for variations of functional substitution.

$$X \Rightarrow W/(W\backslash X)\ (>T)$$
$$X \Rightarrow W\backslash(W/Z)\ (<T) \quad (5)$$

Figure 4: Derivation rules for type-raising.

which cannot combine. The equivalent '$\text{I}_{S/(S\backslash NP)}$ $\text{will}_{(S\backslash NP)/VP}$' can compose to become S/VP.

The derivation rules are provided in Figure 2.1.4. Type-raising is also restricted such that the variable W must be primitive.

While the T combinator could be implemented as a unary rule, implementing it as a binary rule makes it easier to ensure that a type is raised only when there is some other category adjacent with which it could combine – it still must be handled slightly differently to the other combinators, however, as the resulting edge spans only the primary functor, rather than both the functor and the argument.

2.1.5 Conjunction

There are a variety of rules used for conjunction in the literature. The approach (conceptually) is to use a ternary combinator (generally denoted Φ. However, the convention adopted in [4] merely treats and as a lexical entry, but restrict it to combine only through application. This is denoted as '$\text{and}_{X\backslash *X/*X}$', where X is a variable. This is the approach adopted here.

2.2 Parsing

The algorithm used for performing the parsing is the Cocke-Younger-Kasami algorithm, described for CCGs in [3].

Essentially, it constructs every possible edge from pairs of edges spanning 2 leaves, then 3 leaves, and continues until all edges spanning the whole sentence have been constructed. The resulting parse trees can then be read off the chart.

3 Implementation

The parser is implemented according to the NLTK ParseI interface. The CCGChartParser class is defined in the *ccgparser.py* module. Functions for manipulating lexicons are provided in the *ccglexicon.py* module.

The simplest way to understand the interface is most likely to examine code in *demonstration.py*. This gives an example of constructing a lexicon, creating a parser and using it to display derivations for a sentence.

3.1 Lexicon

In order to construct a CCG parser, the first thing that is needed is a lexicon. Rather than attempting to manipulate the complex xml-based format used by the openccg[1] project, a somewhat more intuitive format is presented, based on the format used for defining CFGs.

```
lexicon = ccglexicon.parseLexicon('''
   # Define the set of primitive categories
   # S is the target primitive for the parser
   :- S, N, NP, VP

   # Define a family of words
   Det :: NP/N
   Pro :: NP

   # Define a mapping from a word to a category
   I => Pro
   eat => VP/NP
   and => var\\.,var/.,var
   ...
   ...
   ''')
```

As the slash-types presented in [4] are unsuited to representation in ASCII, the notation used here is that slashes marked with '.' cannot permute, and those marked with ',' cannot compose.

A more thorough example lexicon is provided in *demonstration.py*. The file *ccglexicon.py* also provides a very minimal lexicon derived from the openccg 'tiny-tiny' grammar, which demonstrates subcategories for plurality.

3.2 Rule Sets

A number of example sets of combinators are defined in *ccgparser.py*.

- **ApplicationRuleSet**: $<, >$
- **CompositionRuleSet**: $> B$, $< B$, $< B\times$
- **SubstitutionRuleSet**: $> S$, $< S\times$
- **TypeRaiseRuleSet**: $> T$, $< T$
- **DefaultRuleSet**: All of the above.

It should be reasonably simple to construct different sets, however.

3.3 Parser

Constructing and using the parser is then just like the other NLTK parsers:

[1] http://openccg.sourceforge.net

```
parser = ccgparser.CCGChartParser(lexicon,ccgparser.DefaultRuleSet)

for parse in parser.nbest_parse("I might cook and eat the bacon".split()):
      ccgparser.printCCGDerivation(parse)
```

3.4 Source Files

For reference, the source files provided are:

- *ccgcateg.py*: Classes for handling categories
- *ccglexicon.py*: Code for parsing and handling lexicons
- *ccgparser.py*: Classes for the parser and chart rules
- *combinator.py*: Rules and classes for the combinators.
- *demonstration.py*: An example of how to construct a lexicon and parser.
- *rel_clause.py*: Example code for parsing Wh-relative clauses.
- *testconj.py*: Testing behaviour of conjunctions.

4 Results

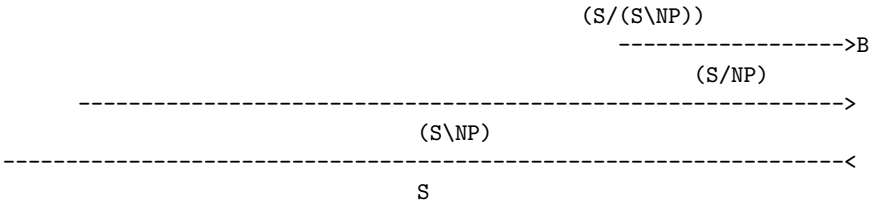
The parser appears to be able to easily handle a variety of interesting linguistic phenomena.

4.1 Wh-relative clauses

The standard rule-set is perfectly capable of handling relative clauses. The only unusual part is the annotation of 'which' with the category $((N\backslash N)/(S/NP))$.

```
you    prefer      that   cake
NP   ((S\NP)/NP)  (NP/N)   N
                 -------------->
                      NP
    --------------------------->
              (S\NP)
--------------------------------<
              S
```

```
that      is        the    cake      which       you    prefer
 NP   ((S\NP)/NP)  (NP/N)   N    ((N\N)/(S/NP))  NP   ((S\NP)/NP)
     ---------------------->B
          ((S\NP)/N)
                        ------>T
                      (N/(N\N))
     -------------------------->B
           ((S\NP)/(N\N))
     ------------------------------------------>B
                 ((S\NP)/(S/NP))
                                             ----->T
```



A number of other examples have been constructed, however are not presented here purely due to space constraints, particularly since longer sentences tend to produce very wide derivation diagrams. Some of them have been included in *rel_clause.py*.

4.2 Subject Extraction

A related problem (and indeed, the motivating example for the S combinator) is handling Wh-relative clauses in the presence of adverbials.

For example, without the S combinators, the sentence 'that is the dough which you will eat without cooking', although very similar, is impossible to parse. The example code is also provided in *rel_clause.py*.

```
Subject Extraction
==================
Without Substitution:

With Substitution:
 that      is        the    dough      which       you     will        eat          without           cooking
  NP   ((S\NP)/NP)  (NP/N)    N    ((N\N)/(S/NP))  NP   ((S\NP)/VP)  (VP/NP)  ((VP\VP)/VP['ing'])  (VP['ing']/NP)
      -------------------->B
           ((S\NP)/N)
                        ------->T
                       (N/(N\N))
      --------------------------->B
            ((S\NP)/(N\N))
                                      ----->T
                                   (S/(S\NP))
                                      ----------------->B
                                            (S/VP)
                                                                  ---------------------------------------->B
                                                                                ((VP\VP)/NP)
                                                                ------------------------------------------<Sx
                                                                                (VP/NP)
                                      ------------------------------------------------------------------->B
                                                                             (S/NP)
                                  ---------------------------------------------------------------------------->
                                                                          (N\N)
      ---------------------------------------------------------------------------------------------------------->
                                                (S\NP)
------------------------------------------------------------------------------------------------------------------<
                                                S
```

5 Evaluation

The constructed parser seems to behave correctly, and also appears to be able to parse some complex sentence structures. Given that the task was to construct a simple parser for exploration of combinatory categorial grammars, it appears to succeed admirably.

The parser does, however, have a number of minor shortfalls, noted below.

5.1 Supertagging

At present, it is necessary to construct a lexicon for the parser by hand. This is clearly infeasible for working with large corpora; instead, modern wide-coverage parsers use a technique called 'supertagging'[1] to automatically annotate each word with the corresponding category.

Given a supertagger, it should be relatively straightforward to modify the parser to initialize leaf edges from the tagger, rather than the lexicon.

5.2 Spurious Ambiguity

One problem with the parser is that it returns a large number of somewhat redundant derivations. The phrase 'spurious ambiguity' is somewhat of a misnomer, because each derivation is indeed valid; however, many of the derivations are semantically equivalent. For example, even for a sentence fragment as short as 'the brown dog', there are multiple derivations:

```
  the    brown  dog
 (NP/N)  (N/N)   N
--------------->B
    (NP/N)
-------------------->
         NP

  the    brown  dog
 (NP/N)  (N/N)   N
        ------------>
             N
-------------------->
         NP
```

The solution to this problem is to keep track of the normalized logical form of the derivation, and add an edge to the chart only if this is unique, rather than the derivation itself.

5.3 Morphological Subcategories

The parser as it stands has rather limited support for subcategorization. This is most obvious in the *openccg_tinytiny* grammar fragment, provided in *ccglexicon.py*. The modal 'the' has two lines in the lexicon – one for singular nouns, and one for plurals.

A better approach would perhaps to allow variables as subcategories, somewhat like the approach used for conjunctions, allowing subcategories to be inherited from arguments upon application.

References

[1] Stephen Clark and James R. Curran. The importance of supertagging for wide-coverage ccg parsing. In *COLING '04: Proceedings of the 20th international conference on Computational Linguistics*, pages 282–288, 2004.

[2] Mark Steedman. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5(3):403–439, 1987.

[3] Mark Steedman. *The Syntactic Process*. The MIT Press, 2000.

[4] Mark Steedman and Jason Baldridge. Combinatory categorial grammar. Draft 5.0, 2007.