A Meta-grammar for CCG

Mark FOREMAN
Dept. of Electrical and Electronic Eng.
The University of Adelaide
Adelaide SA 5005
Mark.Foreman@csiro.au

Daniel McMICHAEL
Information Enhancement Group
CSIRO ICT Centre
Waite Road, Urrbrae SA 5064
Daniel.McMichael@csiro.au

Abstract

Applying CCG to domains outside of linguistics could require different sets of combinators to be developed for each domain. The meta-grammar described in this paper aims to assist such development by enabling simple, succinct expression of both existing and new combinator definitions. It favours the development of an easily-configurable, one-time-coded module that can perform CCG combinations for any combinator set of the researcher's choosing. A preliminary implementation shows both the feasibility and potential of the meta-grammar.

1 Introduction

The merits of Combinatory Categorial Grammar (CCG) have been established via natural language parser implementations like that of Hockenmaier and Steedman (2002) and Clark and Curran (2004). But recent findings show that categorial grammars based on CCG also display promise in domains outside of linguistics (McMichael et al., 2004). Over the years, new combinators have been developed to extend the system of pure categorial grammar (Steedman 2003), but although the set of combinators for CCG seems to have stabilised, this same set may not necessarily be applicable to analyses in other domains. In fact, McMichael et al. introduce two combinators – *functional application* and *modificational application*, both defined in section 2.1 – that are not part of the existing set of CCG combinators. Additionally, functional application cannot even be cleanly defined via a traditional combinator pattern. It is partly this inability of the existing techniques to cleanly define new combinators that motivates the proposal of a meta-grammar for combinator specification.

In this paper, we explain the motivations for and give a specification of the meta-grammar, along with complete examples of how it applies to new and existing combinators. Lastly, we examine the workings and potential of a preliminary implementation. The remainder of this section, however, presents a brief introduction to CCG.

1.1 A Practical Introduction to CCG

CCG operates by first assigning a syntactic category to each word in the sentence, as will be demonstrated in the proceeding example borrowed from Hockenmaier (2003). At this point, the notation for describing categories should be observed. Assuming the simplistic subject-verb-object (SVO) pattern for English, the phrase "buys shares" will form a complete sentence S if it is preceded by a noun phrase, and we write this as:

```
buys shares ├ S\NP
```

So the phrase "buys shares" can be thought of as a function that takes a noun phrase NP as an argument to its left and returns a sentence S.

Furthermore, "buys" will form a S\NP if it is followed by a noun phrase, and this is denoted as:

```
buys ├ (S\NP)/NP
```

In doing this, we have eliminated the need for a separate verb category V, leaving us with the following category assignments:

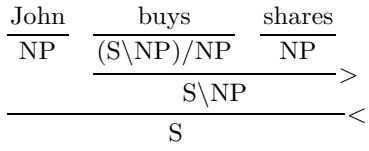
```
John ├ NP
buys ├ (S\NP)/NP
shares ├ NP
```

Formally, a category may be either atomic (S, NP, etc) or complex (S\S, (S\NP)/NP), etc). Complex categories take the general form α/β or α\β, where α and β are themselves categories.

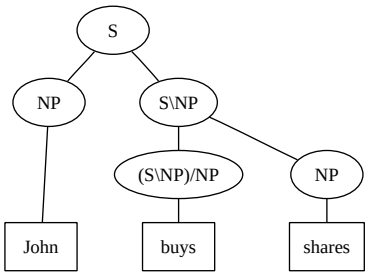
Given the above category assignments, a derivation proceeds as follows: "buys" is combined with "shares" under the operation of *forward application* (the term *forward* referring to both the direction of the slash). The phrase "buys shares" is combined with "John" under the operation of *backward application*. The combinators (operators) that govern these two operations are defined as follows:

X/Y Y ⇒> X
Y X\Y ⇒< X

where X and Y represent any category. Typically, a derivation is represented in the following manner:



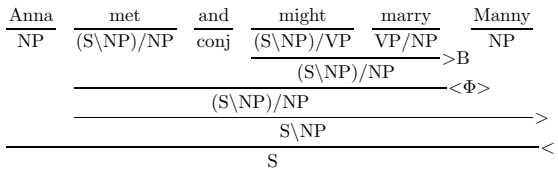
which corresponds to the following tree:

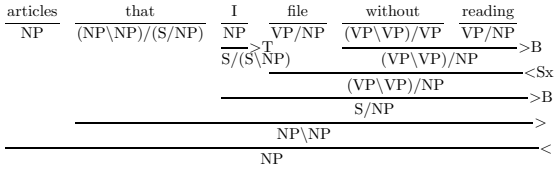


Several other combinators are defined by Steedman (2000) for capturing long-range dependencies in the English and Dutch languages: coordination (Φ) type-raising (T), composition (B) and substitution (S). These combinator families are listed below:

X conj X	⇒<Φ>	X
X	⇒>T	Y/(Y\X)
X	⇒<T	Y\(Y/X)
X/Y Y/Z	⇒>B	X/Z
X/Y Y\Z	⇒>Bx	X\Z
Y\Z X\Y	⇒<B	X\Z
Y/Z X\Y	⇒<Bx	X/Z
(X/Y)/Z Y/Z	⇒>S	X/Z
(X/Y)\Z Y\Z	⇒>Sx	X\Z
Y\Z (X\Y)\Z	⇒<S	X\Z
Y/Z (X\Y)/Z	⇒<Sx	X/Z

The usage of some of these combinators is shown below, using example derivations taken from Steedman (2000) and Hockenmaier (2003).





For further reference on CCG, the reader is directed to Steedman (1996) and (2000).

1.2 An Historical Note on CCG

The slash notation seen in categories of CCG stems from that used in the early works on pure categorial grammar by Ajdukiewicz (1935), Bar-Hillel (1953) and Lambek (1958). Steedman (1993) explains that he and Dowty refined these earlier notations, leading to the more consistent and readable style that is described in section 1.1. The only rules permitted by pure categorial grammar are forward and backward application. CCG extends this system with the above-listed rules, based on Curry and Feys' *combinators* – a term coined in their 1958 work on combinatory logic, where they examined devices that operate on functions, irrespective of the number of arguments. The combinatory nature of CCG rules enables a transparent mapping between syntactic and semantic form, thus providing one of the major appeals of this grammar formalism.

2 Motivations

As can be seen in section 1.1, enumerating the entire set of combinators can be lengthy. Given that combinators in a family like {>B, >Bx, <B, <Bx} differ only by the direction of the slashes and order of the operands, it seems wasteful to present each one explicitly. A more compact representation is afforded by specifying only the pattern for >B, along with the transformations required to obtain the variations >Bx, <B and <Bx. The proposed meta-grammar provides a method for succinctly specifying these variations.

Without recognising the similarity within combinator families, and even between combinator families, writing code to apply combinators can be laborious, error-prone and wasteful, unless these similarities are exploited for optimum code reuse. The implementation of this meta-grammar takes full advantage of intra- and inter-family similarities.

2.1 New Domains

Providing a meta-grammar by which to specify combinator families lends itself to a single-module implementation that can be easily configured to handle new combinators. This capability is important because, although the set of combinators for use in linguistics seems to have matured, there

are other domains that stand to benefit from CCG analyses, but for which grammar development is still in its infancy. In particular, the authors are currently developing a generic parser capable of being configured to specific domains, including, but not limited to, NLP and situation assessment. Research into applying CCG to these domains is being assisted by an ability to perform rapid prototyping on their grammars.

The two new combinators mentioned in the introduction – functional application (F) and modificational application (M) – are defined as follows:

$$X/Y \quad Y \Rightarrow_{>F} X$$
$$Y \quad X\backslash Y \Rightarrow_{<F} X$$
$$X/X \quad X \Rightarrow_{>M} X$$
$$X \quad X\backslash X \Rightarrow_{<M} X$$

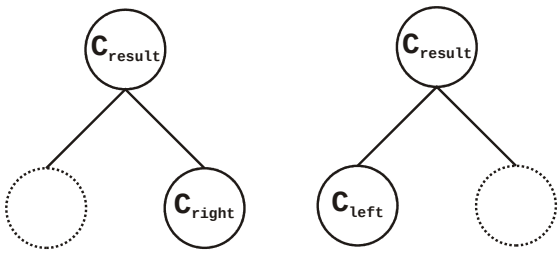
with the caveat that X≠Y. These combinators find use in the authors' research in both English parsing and situation assessment. Note that >F and >M together cover all combinations possible under the traditional vanilla forward application rule >; similarly for <F and <M with <. It will suffice to say here that the reason for splitting the traditional rules into two was to correctly handle head inheritance while maintaining a simple mathematical model; further explanation is not within the scope of this paper[1]. The >F and <F combinators defined above would incorrectly be interpreted as the vanilla > and < combinators if it were not for the caveat. So this example not only demonstrates new combinators, but also highlights the shortcomings of the current methods for specifying combinators. The proposed meta-grammar provides an elegant (caveat-free) solution to this problem.

Another domain that is planned for investigation is geology. The general intent is to analyse vertical sequences of discrete sedimentary layers in a manner analogous to English parsing, where sequences of (discrete) words are analysed. Griffiths (1989) founded the precursor to this research by demonstrating that meaningful analyses could be performed on sedimentary sequences using a context-free grammar. He also speculated that context-sensitive analyses might be able to resolve some ambiguities, lending weight to the application of CCG, which is mildly context-sensitive, to such a task.

[1] The authors would like to credit the work of Geoff Jarrad in developing these two combinators.

2.2 Converting Treebanks

When employing a statistical parser, a suitable corpus of pre-parsed sentences is required for training the probabilities. However, altering the grammar through the addition or deletion of combinators (as is done when applying CCG to new domains) requires a new corpus to be marked-up accordingly. This process typically requires converting the context-free derivation trees from the Penn Treebank (Marcus et al., 1993, 1994) to intermediate binary context-free trees and then finally to CCG trees. The second stage of conversion (binary CF to CCG) requires a technique referred to as *inverse combination*, where an unknown left or right category is determined given the result category. This contrasts to regular combination, where the unknown result is deduced from two known operands. There are two types of inverse combination: *missing-left* (when the right operand and result are known) and *missing-right* (when the left operand and result are known. Missing-left and missing-right scenarios are shown left and right respectively below.



It will be shown in section 4.2 that a standard implementation of the meta-grammar can be made to perform these operations by merely permuting some of the configuration information passed to the module.

2.3 Python Implementation

The implementation described in this paper was coded in Python (Python Programming Language, 2004). Python was chosen for its ability to aid rapid prototyping and for its ease of integration with much faster C code. Thus we hope to benefit from lower coding times and easier debugging, with the option to port to C and re-integrate any mature code that is deemed time-critical.

3 The Meta-grammar

This section details the meta-grammar that controls the specification of a CCG. The set of combinators are specified as a list of combinator templates, one template for each combinator family:

```
COMBINATOR-SET :=
```

COMBINATOR-TEMPLATE$_1$
…
…
COMBINATOR-TEMPLATE$_m$

Each combinator template defined separately, as well as any atomic variations referenced in the templates. These are both described below, and proceeded by some example templates.

It is worth noting that this proposal focuses primarily on specifying combinators for the express purpose of performing combinations. The corresponding semantics (logical forms) may be associated with operands and the result, following from Steedman (2000).

3.1 Specifying a Combinator Template

A combinator template is specified as a tuple:

```
COMBINATOR-TEMPLATE :=
  ( TYPE,
    OPERAND-PATTERN-LIST,
    RESULT-PATTERN,
    PERMITTED-VARIATIONS )
```

where the entries in the tuple are defined as:

TYPE: an identifier for the combinator that should be unique across all other combinator templates. Typically it is a single character; in section 1.1 we saw them as Φ, T, B and S.

OPERAND-PATTERN-LIST: the ordered list of n operand patterns. Typically n=2 since most combinators are binary operations, although n=1 for type-raising (T). The syntax for these patterns is given in section 3.2.

RESULT-PATTERN: a pattern that specifies how to construct the resulting category from operand categories that successfully match the operand patterns.

PERMITTED-VARIATIONS: as mentioned in section 2, only one pattern set is specified per family of combinators. Each variation in the family is specified as a tersely coded entry in this list. For instance, the composition family (B) would have permitted-variations = { >, >x, <, <x }.

3.2 Specifying a Pattern

The patterns specified in the combinator template must conform to the following EBNF syntax:

```
<PATTERN> := <ATOMIC>|
   <COMPOUND>
<ATOMIC> := <A>['e'|'n']<N>
<A> := ('A'|…|'Z')+
<N> := ('0'|…|'9')+
<COMPOUND> := <LEFT>
   (<RIGHT>|'['<RIGHT>']')
<LEFT> := <ATOMIC>|
   '('<COMPOUND>')'
<RIGHT> := <SLASH><LEFT>
<SLASH> := ('\'|'/')<N>
```

For simplicity of expression, we introduce the semantic requirement that an <A>n<N> pattern may only occur immediately after a slash. Alternatively, we could provide a completely context-free grammar for patterns through a slightly less intuitive EBNF, by redefining <ATOMIC> and <RIGHT>:

```
<ATOMIC> := <A>['e']<N>
<RIGHT> := <SLASH>
   (<LEFT>|<A>'n'<N>)
```

Some patterns that conform to this syntax include:

X_1
$Y_1/_2Y_3$
$(X_1/_1Y_1)\backslash_2X_2$

Atomic patterns (patterns without slashes or brackets) are specified as alphanumeric strings which to allows for greater control over pattern specification. Any two atomic patterns (A'N' and A″N″) and the categories they match (C' and C″ respectively) are governed by the following constraints:

A'=A″, N'=N″ ⇒ C'=C″
A'=A″, N'≠N″ ⇒ C'≠C″

As an example, suppose we want to match some category to the pattern $((X_1/_1X_1)/_2X_2)/_3Y_1$, then the subcategory in the position of the first X_1 must be equal to the subcategory in the position of the second X_1, but must be *distinct* from the subcategory in the position of the X_2 (and any other $X_{<N>}$ that might have been in the pattern). The subcategories in the positions of X_1, X_1 and X_2 are *independent* of the subcategory in the position of Y_1. For example, this pattern would match the categories ((A/A)/B)/A, ((A/A)/B)/B and ((A/A)/B)/C, but not ((A/C)/B)/A or ((A/A)/A)/A.

The presence of an 'e' in an atomic pattern indicates that the atomic pattern will only match with an atomic category. Thus X^e_1 will match category A, but not A/A.

The presence of an 'n' in an atomic pattern indicates that the atomic pattern will allow matching to an unlimited number of arguments, similar to the "$ convention" described in

(Steedman 2000). A pattern $X_1/_1Y^n_1$ would match categories A/B, (A/B)/C, ((A/B)/C)/D, etc.

Square brackets (if present) in a pattern surround an optional portion of that pattern. For example, the pattern $X^e_1[/_1Y^e_1]$ would match categories A and A/B, but not (A/B)/C or A/(B/C).

3.3 Specifying Variations

Each combinator in a given family corresponds to exactly one variation in the permitted-variations list of that family's combinator-template. Suppose we have operand and result patterns:

```
TYPE = B
OPERANDS = X1/1Y1, Y1/2Z1
RESULT   = X1/3Z1
```

then a > in the permitted-variations list corresponds to the combinator:

$X_1/Y_1\ Y_1/Z_1 \Rightarrow_{>B} X_1/Z_1$

That is, forward combination > does not alter slash directions or operand order. On the other hand, backward combination < reverses all slashes and operand order, so a < in the permitted-variations list would correspond to the combinator:

$Y_1\backslash Z_1\ X_1\backslash Y_1 \Rightarrow_{<B} X_1\backslash Z_1$

Other atomic variations may be defined and used with either < or >. An atomic variation that is used in generating the composition (B) family is:

$x:\{/_2,/_3\}$

That is, the x variant reverses the direction of slash 2 and slash 3. The effect of atomic variations is successive. So a variation like <x would have operands and all slashes reversed by <, but the x would reverse slashes 2 and 3 back to their original orientation (in this case, forward):

$Y_1/Z_1\ X_1\backslash Y_1 \Rightarrow_{<Bx} X_1/Z_1$

Taking this one step further, suppose we invent an arbitrary variant $i:\{/_3\}$, then the combinator corresponding to variant <xi would be:

$Y_1/Z_1\ X_1\backslash Y_1 \Rightarrow_{<Bxi} X_1\backslash Z_1$

Slash 3 has been reversed once by <, again by x and again by i, giving an overall effect of a single reversal.

3.4 Some Example Templates

This section is a simple demonstration of how the above-described meta-grammar can be used to specify both existing (type-raising, composition) and new (functional application, modificational application) combinators.

3.4.1 Type Raising

```
TYPE = T
OPERANDS = X1
RESULT = Y1/1(Y1\2X1)
VARIATIONS = {>,<}
```

$X_1 \Rightarrow_{>T} Y_1/(Y_1\backslash X_1)$
$X_1 \Rightarrow_{<T} Y_1\backslash(Y_1/X_1)$

3.4.2 Composition

Note that this template specifies general composition (B^n).

```
TYPE = B
OPERANDS = X1/1Y1, Y1/2Z^n1
RESULT = X1/3Z^n1
VARIATIONS = {>,>x,<,<x}
```

$X_1/Y_1\ Y_1/Z^n_1 \Rightarrow_{>B} X_1/Z^n_1$
$X_1/Y_1\ Y_1\backslash Z^n_1 \Rightarrow_{>Bx} X_1\backslash Z^n_1$
$Y_1\backslash Z^n_1\ X_1\backslash Y_1 \Rightarrow_{<B} X_1\backslash Z^n_1$
$Y_1/Z^n_1\ X_1\backslash Y_1 \Rightarrow_{<Bx} X_1/Z^n_1$

3.4.3 Functional Application

```
TYPE = F
OPERANDS = X1/1X2, X2
RESULT = X1
VARIATIONS = {>,<}
```

$X_1/X_2\ X_2 \Rightarrow_{>F} X_1$
$X_2\ X_1\backslash X_2 \Rightarrow_{>F} X_1$

3.4.4 Modificational Application

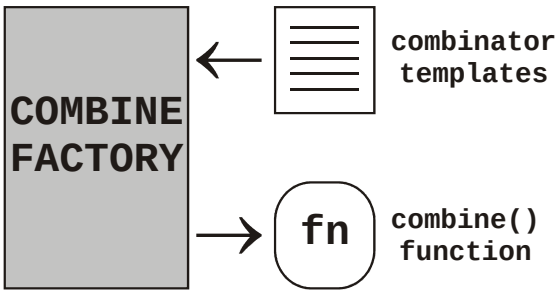
```
TYPE = M
OPERANDS = X1/1X1, X1
RESULT = X1
VARIATIONS = {>,<}
```

$X_1/X_1\ X_1 \Rightarrow_{>F} X_1$
$X_1\ X_1\backslash X_1 \Rightarrow_{>F} X_1$

4 Using the Implementation

A prototype module has been developed in Python to implement the meta-grammar described in this paper. The module can be thought of as a factory which takes a combinator-set conforming to the

definition in section 3 and returns a single function, `combine()`.



The `combine()` function takes any number of operand categories as arguments and returns a list of (result-category, combinator) tuples that corresponds to all possible categories that can be derived from the input categories.

4.1 Combination

As an example, let us consider the module when configured by the type-raising and composition combinator templates given in sections 3.1 and 3.2. The input to the `combine()` function is a sequence of operand categories, and the output is a list of possible resulting categories and their corresponding combinators.

Suppose the input is a pair of categories, `A/B` and `B\C`. Type-raising is immediately discounted by the function because it is unary and thus cannot operate on a pair of categories. Consequently, the function only considers the composition (B) combinators. The function attempts to match the first category, `A/B`, with the first operand pattern, $X_1/_1Y_1$, and the second category, `B\C`, with the second operand pattern, $Y_1/_2Z^n_1$, ignoring slash directions for the moment. This match is successful, and results in a *match dictionary* of $\{X_1{:}A,\ Y_1{:}B,\ Z^n_1{:}C\}$. The slashes are then found to match those required for forward crossing (`>x`) composition $\{s_1{:}/,\ s_2{:}\backslash\}$, but not for vanilla forward composition $\{s_1{:}/,\ s_2{:}/\}$. From these matches, the result can be built: $X_1\backslash Z^n_1{:}A\backslash C$. To attempt the backwards combinations, the function then tries to match the input categories to the reversed sequence of operand patterns, i.e. `A/B` with $Y_1/_2Z^n_1$ and `B\C` with $X_1/_1Y_1$. This attempt fails because $Y_1{=}A$ in the first category, while $Y_1{=}C$ in the second category. So for the input `A/B B\C`, the output is a single category-combinator pair: `(A\C >Bx)`.

If the input were `A/B C`, the output would be an empty list since the second category `C` will not match the structure of either of the operand patterns.

Suppose now the input is a single category `A`. The composition combinators can be immediately discounted since they require two operands. However, it does match the single operand pattern for the type raising combinator, giving match dictionary $\{X_1{:}A\}$. This conforms to both the forward and backward type-raising, so the function would output a pair list `{(*/(*\A) >T), (*\(*/A) <T)}`, where the `*` character indicates that there was no match for Y_1 in the input. The handling of these wildcards rests with the client software.

4.2 Inverse Combination

A very useful property of the meta-grammar and its associated implementation is that it can be configured to deduce a child category given the derived category and the other child/children. A process we term *inverse combination*.

Consider the case of a binary combinator with pattern:

```
OPERANDS = A B
RESULT   =     C
VARIATIONS = {>*,<*}
```

where `A`, `B`, `C` are pattern *placeholders* (– they are obviously not valid patterns themselves), `>*` represents some number of forward variations and `<*` represents some number of backward variations. Now suppose we know the left and result categories (c_A and c_C), and wish to enumerate all valid right categories (c_B) – the *missing-right* scenario. This is achieved via a two-step process, involving the instantiation of two `combine()` functions:

```
OPERANDS = A C
RESULT   =     B
VARIATIONS = {>*}
    ↓
[COMBINE FACTORY]
    ↓
combine1()

OPERANDS = C A
RESULT   =     B
VARIATIONS = {<*}
    ↓
[COMBINE FACTORY]
    ↓
combine2()
```

Simple addition of the two returned lists obtains the desired result:

$$combine_1(c_A, c_C) + combine_2(c_A, c_C)$$

This works because $\texttt{combine}_1()$ returns the list of c_B's that result from valid matches of $\texttt{A:}c_A$, $\texttt{C:}c_C$. $\texttt{combine}_2()$ also matches $\texttt{A:}c_A$, $\texttt{C:}c_C$, but its operation is a little less obvious: because $\texttt{combine}_2()$ only considers backward combinations, it always reverses the order of its operands c_A and c_C, so A is still compared with c_A, and C with c_C.

Inverse combination for the *missing-left* scenario can be performed similarly, so in the interest of brevity its detail is omitted.

While the above approach may seem awkward, keep in mind that no changes are required to the meta-grammar definition or to the implementation's code base. So inverse combination is obtained for free.

5 Conclusion

We have defined a meta-grammar for specifying complete families of CCG combinators. This meta-grammar covers existing combinators, but more importantly, it provides a guide for specifying and using new combinators. A brief look at a preliminary implementation reveals that the meta-grammar is indeed practical, and lends itself to powerful exploitation.

6 Acknowledgements

This paper is based on work supported by Boeing and CSIRO, and by a Commonwealth scholarship (APA) plus Woodside PhD top-up scholarship to the first author. The authors would like to thank Geoff Jarrad for his input to discussions and assistance with reviewing. Many thanks also to the official reviewers for their helpful suggestions.

References

(2004). Python Programming Language, Python Software Foundation. 2004.

Ajdukiewicz, K. (1935). "Die syntaktische Konnexität." *Studia Philosophica* 1: 1-27.

Bar-Hillel, Y. (1953). "A Quasi-Arithmetical Notation for Syntactic Description." *Language* 29: 47-58.

Clark, S. and J. Curran (2004). Parsing the WSJ using CCG and Log-Linear Models. *42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, Barcelona, Spain.

Curry, H. and R. Feys (1958). Combinatory logic. Amsterdam, North-Holland.

Griffiths, C. M. (1989). "The nature of the geological representation language and consequent constraints on machine interpretation." *Advances in Geophysical Data Processing* 3: 49-77.

Hockenmaier, J. (2003). Data and Models for Statistical Parsing with Combinatory Categorial Grammar. School of Informatics. Edinburgh, University of Edinburgh: 280.

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

Lambek, J. (1958). "The mathematics of sentence structure." *American Mathematical Monthly* 65: 154-170.

Marcus, M. P., G. Kim, et al. (1994). The Penn treebank: Annotating predicate argument structure. *Human Language Technology Workshop*.

Marcus, M. P., B. Santorini, et al. (1993). "Building a Large Annotated Corpus of English: The Penn Treebank." *Computational Linguistics* 19(2): 313-330.

McMichael, D., G. Jarrad, et al. (2004). Modelling, Simulation and Estimation of Situation Histories. *7th International Conference on Information Fusion (Fusion 2004)*, Stockholm, Sweden, International Society for Information Fusion.

Steedman, M. (1993). "Categorial grammar." *Lingua* 90(3): 221-258.

Steedman, M. (1996). Surface Structure and Interpretation. Massachusetts, MIT Press.

Steedman, M. (2000). The Syntactic Process. Massachusetts, MIT Press.

Steedman, M. and J. Baldridge (2003). Combinatory Categorial Grammar (Draft 4.0, August 10, 2003). ftp://ftp.cogsci.ed.ac.uk/pub/steedman/ccg/manifesto.pdf [Internet]. Accessed 15 September 2004.