HIGH SCHOOL EDITION
Journal of Student Research
Volume 12 Issue 1 (2023)

From Combinatory Categorial Grammar to the Calculus of Constructions: A Review of ccg2lambda.

Yeonho Jung[1] and Daniel Briggs[#]

[1]Peddie School, Hightstown, NJ, USA
[#]Advisor

ABSTRACT

The following is an overview of the concepts and systems necessary to build an understanding of the framework from *ccg2lambda: A Compositional Semantics System*, which produces logical semantic representations of natural language sentences. These representations are used in tandem with a theorem prover to conduct inferences. But, to the reader unfamiliar with the various tools within computer science, linguistics, and mathematics that were used to construct ccg2lambda and its dependencies, it can be difficult to understand the inner workings of the framework. Thus, an explication of these concepts could be useful for fostering further development. Concepts such as mathematical logic, combinatory categorial grammar, the calculus of constructions, and interactive theorem provers are covered, as well as the two systems C&C and ccg2lambda. A functional version of the ccg2lambda framework can be found at https://github.com/ajung202/ccg2lambda.

Introduction

Logic is a field in both philosophy and mathematics that deals with determining valid ways to deduce conclusions. Over the course of the past two centuries, logic has developed greatly, coming to encompass numerous branches from propositional logic to first-order logic and higher-order logic, each of which has a different level of expressiveness. The current work fosters understanding of existing progress on natural language processing, specifically in its application of logic. Ultimately, this aims to advance application of logic in everyday life.

One way to do so is via the meaning representations given by the *ccg2lambda* framework together with existing interactive theorem provers such as Coq. To produce meaning representations from natural language, the framework implements the Clark & Curran parser in tandem with manually defined semantic templates, which are central to the *ccg2lambda* system. Using these meaning representations, a script written using the interactive theorem prover Coq attempts to determine whether given conclusions are deducible from given sets of premises. The deductive ability of the framework was tested on the FraCaS textual inference problem set. The theorem prover uses the calculus of constructions, which is derived from type theories and specifically the simply-typed lambda calculus.

Because of the interdisciplinary nature of this endeavor, which involves philosophy, mathematics, linguistics, and computer science, it can be difficult for readers to fully understand and thus also to further progress. Hence, a detailing of the theory behind each component of the *ccg2lambda* framework and its background is provided.

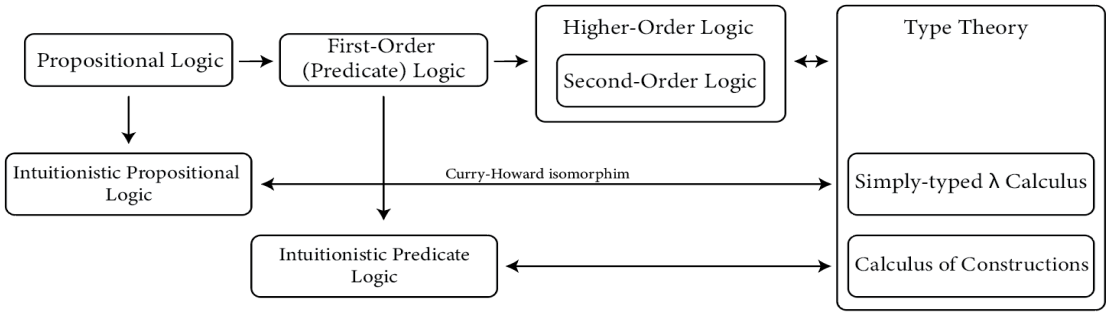

Figure 1. This flowchart provides an overview of relations and dependencies of the components to come.

Components

Propositional Logic

Tracing its origins to stoic philosophers in the 3rd century BC, propositional logic is a formalization of reasoning specifically dealing with propositions (Haaparanta, 2009, p. 3). *Propositions*, usually represented as $\{p, q, r, \dots\}$, are declarative sentences that evaluate to true or false. These propositions are connected via *connectives* $\{\neg, \rightarrow, \wedge, \vee, \leftrightarrow\}$, representing negation and conjunctions (Enderton, 2013, p. 14). The truth value of each propositional formula can be represented through truth tables such as that of Figure 2, which can be used to prove the law of the contrapositive. Proofs in propositional logic have a set of premises and a conclusion. Deductions are made via inference rules such as *modus ponens*, which means that assuming a conditional statement and its hypothesis, its conclusion can be deduced (Enderton, 2013, p. 110). The Fitch-style proof of the law of the contrapositive is provided in Figure 3. The law of the excluded middle, which states that $p \vee \neg p$ is a tautology, can also be proved by either of these methods.

p	q	$\neg p$	$\neg q$	$p \rightarrow q$	$\neg q \rightarrow \neg p$
T	T	F	F	T	T
T	F	F	T	F	F
F	T	T	F	T	T
F	F	T	T	T	T

Figure 2. Example of a truth table. Observe that the right two columns are logically equivalent. Adding a column with expression $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$ has true as all its truth values, demonstrating this equivalence.

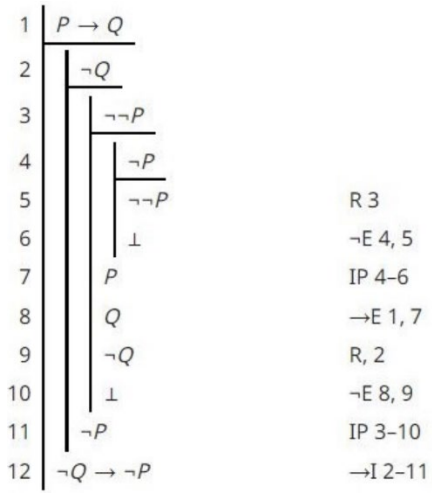

Figure 3. Fitch-style proof of the law of the contrapositive. Modus ponens is used on line 8 of the proof.

First-Order Logic (FOL)

Discovered independently by Frege and Peirce in the late 19th century (Moore, 1988), *first-order logic*, also known as *predicate logic*, is a formalization of reasoning that extends propositional logic: FOL can reason about groups of objects rather than single propositions. FOL has four components. First, like propositional logic, there are connectives for variables $\{\neg, \rightarrow, \wedge, \vee, \leftrightarrow\}$. Second, there are *quantifiers* $\{\forall, \exists\}$, meaning "for all" and "there exists," which provide the scope of a variable. Third, to describe and reason about objects, it implements predicate symbols $\{F, G, H, \ldots\}$ each of which has its own *arity*, the number of variables it takes, and evaluates to either true or false based on their values. Fourth, FOL maps objects via functions of several variables $\{f, g, h, \ldots\}$ (Barwise, 1977). Like proofs in propositional logic, proofs in FOL have a set of premises and a conclusion and can use all the inference rules of propositional logic (Enderton, 2013). In addition, first-order logic proofs can implement the inference rules of universal instantiation, existential generalization, and existential instantiation, which involve the quantifiers.

A *theory* is a set of sentences that are considered to be true such that all implications of those sentences are within the set. A set of *axioms* for a given theory is a subset of the theory that entails (implies) every sentence in the theory.

The first-order theory of the natural numbers (including 0) is usually generated by an infinite set of axioms referred to as the *Peano axioms*, which establish facts about successorship, addition, and multiplication, and contain infinitely many instances of the induction principle. A sentence in the first-order theory of the natural numbers asserting that the remainder of any natural number when divided by 2 is 0 or 1 is shown below:

$$\forall n \exists m \, (n = 2m \vee n = 2m + 1)$$

The literal translation for the above sentence is "for all n, there exists m such that n is equal to $2m$ or n is equal to $2m + 1$."

Second-Order Logic (SOL) and Higher-Order Logic (HOL)

Introduced by Frege in his *Begriffsschrift* in 1879 (Haaparanta, 2009, p. 232), *second-order logic* is a formalization of reasoning that extends first-order logic. While FOL quantifies over elements, SOL quantifies over

both objects and properties (Leivant, 1994). This grants SOL more expressive power. To highlight this, take the law of induction represented in FOL and SOL respectively:

$$\Big(\phi(0) \wedge \forall k\big(\phi(k) \rightarrow \phi(k+1)\big)\Big) \rightarrow \forall k \phi(k)$$

$$\forall \phi \Big(\Big(\phi(0) \wedge \forall k\big(\phi(k) \rightarrow \phi(k+1)\big)\Big) \rightarrow \forall k \phi(k)\Big)$$

On one hand, induction in FOL is a FOL schema for the induction axiom, ϕ being a placeholder for other properties, and thus consists of infinitely many sentences rather than one. On the other hand, induction in SOL is a SOL sentence that encapsulates the induction axiom, quantifying over all properties. Quantification scopes can continue to be increased in this manner, leading to the general term higher-order logic, encompassing all logic beyond first-order logic.

Type theory, discussed below, goes hand-in-hand with higher-order logic, for higher-order logic has elements of different types.

Intuitionistic Logic

First developed by Arend Heyting in 1928 (Haaparanta, 2009), *intuitionistic logic*, unlike standard logic, does not have the law of double negation elimination or the law of the excluded middle as rules of inference. (Bezhanishvili & de Jongh, 2006). Double negation elimination is demonstrated in line 7 of Figure 3.

By not implementing these two inference rules, one obtains intuitionistic propositional logic from propositional logic or intuitionistic predicate logic from predicate logic. This is a massive difference, for it removes the ability to perform proof by contradiction. Though David Hilbert stated that the lack of the law of excluded middle to be "tantamount to relinquishing the science of mathematics altogether," intuitionistic logic provides great value in proofs involving the existence property—intuitionistic logic can be paired with proof assistants like Coq to iterate many cases. (Nozick, 1996).

Lambda Calculus

Invented by Alonzo Church in 1928 (Cardone & Hindley, 2006) dealing with foundations of mathematics, *lambda calculus* is a formal system in mathematical logic representing computation through lambda terms. Lambda terms can be constructed from variables, abstractions, and application: variables represent parameters; abstractions define functions; application applies a function to parameters in a left-associative manner (Barendregt, 1984). Here is an example:

$$\lambda x.F = 2^x + 1$$

$$(F\ 2)$$

The first equation shown is an abstraction in which $F = 2^x + 1$ binds the variable x to F, taking in x and returning $2^x + 1$. Lambda calculus terms that are constructed via variables, abstractions, and applications can be reduced via operations such as *β-reduction*, which evaluates terms defined by function application. For instance, the β-reduction of the application of F given above is shown below:

$$(F\ 2) = 2^2 + 1 = 5$$

Type Theory

First proposed by Bertrand Russell to avoid Russell's paradox, *type theory* offers formalized representations for *type systems*, logical systems that delineate different terms into different types (Bell, 2012). For instance, in a type system containing a natural number type *nat* and a character type *char*, the following expressions indicate the type of each of the terms 1, 10, a, and d:

$$1\text{: nat}$$
$$10\text{: nat}$$
$$\text{a: char}$$
$$\text{d: char}$$

In any type system, *functions* are terms that can either be defined by rule or expressed as lambda terms. *Lambda terms* are constructed to take in given inputs of a certain type and to return outputs of a certain type. Lambda terms have the general form (λ variableName : type . term). For instance, $(\lambda\ \text{x} : \text{nat} .\ \text{add}(x\ 1)) : \text{nat} \rightarrow \text{nat}$ would be a function that takes in a natural number x and returns the natural number $\text{x} + 1$. Calling this function, $(\lambda\ \text{x} : \text{nat} .\ \text{add}(x\ 1))\ 10\text{: nat} = (add\ 10\ 1)\text{: nat} = 11\text{: nat}$.

Some commonly defined types are the empty type, unit type, boolean type, and natural numbers. In addition, types dependent on other types such as the sum type and the product type are commonly defined. A type system with ∧ and ∨ types leads to intuitionistic logic.

Simply-Typed Lambda Calculus and the Calculus of Constructions

The *simply typed lambda calculus* is a minimal type theory that only implements the type constructor "→" over some collection of base types. Suppose the base types are natural numbers and the character type. The type constructor "→" would construct function types as follows.

$$\text{F}_1 \coloneqq \text{nat} \rightarrow \text{nat}$$
$$\text{F}_2 \coloneqq \text{nat} \rightarrow \text{char}$$
$$\text{F}_3 \coloneqq \text{char} \rightarrow \text{nat}$$
$$\text{F}_4 \coloneqq \text{char} \rightarrow \text{char}$$

More types can be developed by nesting right arrows. For instance,

$$\text{F}_5 \coloneqq \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$$

Via the Curry-Howard isomorphism, a proof in intuitionistic propositional logic is related to a term in the simply-typed lambda calculus (Church, 1940). Extending upon this relation, a proof in intuitionistic predicate logic can be related to a term in the Calculus of Constructions, which is the foundation of the Coq Theorem Prover (Coquand & Huet, 1988).

Coq Theorem Prover

Created by Thierry Coquand and his team in 1989 (Bertot & Pierre, 2004), Coq is an interactive theorem-proving system that implements the calculus of constructions. Coq allows users to interactively develop machine-checked proofs, which not only eliminates human error from proof verification but also allows immense case-by-case checking to be expedited. One example is the proof of the four-color theorem, which states that the regions of any planar map can be filled with four colors such that no two regions with the same color are adjacent. Initially, it was proved by Appel and Haken in 1976 assisted by computer programs (Appel & Haken,

1986). Later, in 2005, Benjamin Werner and Georges Gonthier formalized a proof with Coq, removing the potential for human error in the earlier proof (Gonthier et al., 2008).

A *goal* is a term that Coq aims to prove. To do so, Coq implements a series of *tactics*, which either prove a goal or replace it with one or more goals. As shown in Figure 4, tactics such as simplify, rewrite, and reflexivity are written in lowercase whereas other keywords are in title case.

The below example proves that the reversal of the list obtained by appending an element *x* to a list *l* is equal to the reversal of *l*, prepended by *x*. This proposition is defined as a lemma in lines 4 and 5. The proof begins on line 7. It performs induction on *l*. On the right, the inductive hypothesis *IHL* is shown as well as the goal below it.

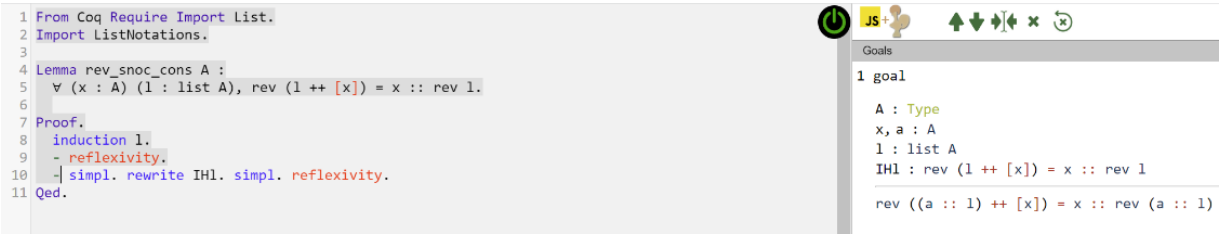

Figure 4. Coq theorem prover run on an internet browser. The highlighted section on the left represents the point in the proof, as shown in the right—there is an unresolved goal below the line.

In ccg2lambda's Coq script, axioms and tactics are defined to resolve terms fed by the ccg2lambda pipeline. For instance, the axiom veridical_true is defined as forall P, (_true P -> P). The tactic solve_veridical_true is a manually defined tactic that can be applied to resolve goals using this axiom. In natural language, P here refers to a dependent clause that follows "It is true that…".

Context-Free Grammar (CFG)

Developed in the mid-1950s by Noam Chomsky, *context-free grammar* defines grammatical structures, as the name suggests, in a context-free way. Context-free grammar constructs a system of non-terminal and terminal symbols. Non-terminal symbols, also called *variables*, eventually resolve to terminal symbols. Non-terminal symbols are resolved by the *production rules*, which replace non-terminal symbols with strings of variables or terminals (Cremers & Ginsburg, 1975). Terminal symbols can be stringed to form *expressions*. S is used to denote the *start symbol*, which must appear alone at the beginning of any derivation. The set of all expressions that can be produced by a given context-free grammar is called a *context-free language*.

Consider a context-free grammar where 1, a, and + are terminal symbols. Take the three derivation rules given below:

$$\begin{aligned} S &\rightarrow S + S \\ S &\rightarrow 1 \\ S &\rightarrow a \end{aligned}$$

Then the expression $1 + a$ can be derived as follows:

$$\begin{aligned} & S \\ \rightarrow\ & S + S \\ \rightarrow\ & 1 + S \\ \rightarrow\ & 1 + a \end{aligned}$$

But CFG does not model natural language well, for natural language is not context free.

Combinatory Categorial Grammar (CCG)

Developed by Mark Steedman starting in the 1980s (Steedman & Baldridge, 2011), *CCG* defines grammar structures by giving different elements such as verbs different syntactic categories and identifying them with functions with directionality of their arguments as well as the type of their result (Steedman, 1996). For instance, take the following example from Steedman, 1996:

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

where the definitions of forward and backward application are denoted as follows:

$$X/Y\ Y \rightarrow X\ (>)$$
$$Y\ X\backslash Y \rightarrow X(<)$$

Here, *likes* is categorized as a function that takes in a noun phrase via forward application and then another noun phrase via backward application. This means that *likes* requires a noun phrase after and before it. Consider the following resolution of the sentence *Mary likes musicals*:

$$\dfrac{\dfrac{\text{Mary}}{NP}\quad\dfrac{\dfrac{\text{likes}}{(S\backslash NP)/NP}\quad\dfrac{\text{musicals}}{NP}}{S\backslash NP}>}{S}<$$

Figure 5. This is an example of a sentence being derived and resolved in CCG. This figure was taken directly from Steedman, 1996.

CCG can produce derivations and dependency structures. *Derivations* are like Figure 5, showing the breakdown of a natural language sentence into CCG over the course of multiple steps of application. A *dependency structure* provides less information—it too shows the breakdown of the sentence into CCG but does not contain levels of application. Sentences that are more complicated can have multiple derivations for a given dependency structure.

CCG outputs are often tagged with feature structures such as [dcl], which stands for declarative sentences.

Clark & Curran Parser (C&C)

Developed by Clark and Curran in 2007, *C&C* is an efficient wide-coverage tool that parses natural language into CCG, which indicates all the syntactic dependencies of a natural language sentence as discussed above (Clark & Curran, 2007). The C&C parser was built via a discriminative log-linear model. The discriminative model is a machine learning model that is fed both correct and incorrect combinatory categorial grammar parses from CCGbank, which consists of sentences and their correct CCG derivations developed by Hockenmaier and Steeedman (Hockenmaier & Steedman, 2007). The log-linear model is a statistical model, assigning probabilities to different syntactic dependencies of C&C parses.

C&C's efficiency in training and parsing is notable. This was enabled by the supertagger, which assigns CCG lexical categories to words. Because there could be multiple derivations for a sentence, the authors used dynamic programming with a CCG chart, which efficiently represents all derivations.

ccg2lambda

The ccg2lambda framework (Mineshma et al., 2015) takes in as syntactic representations CCG derivations given by C&C. It uses around 100 manually defined semantic templates found in its YAML file recursively on the aforementioned tree representation of a sentence. Here is an example of a semantic template for the conjunction *and* when it is used to join two sentences.

category: S\S
rule: conj
semantics: \L S1 S2. (S1 & S2)
child0_surf: and

Semantic features such as [dcl], which stands for declarative sentences, are often tagged on CCG outputs, as mentioned above. Depending on these tags, different semantic templates will be applied. For instance, take:

category: (S[dcl=true]\NP)/(S[to=true]\NP)
semantics: \E V Q. Q(\w.TrueP, \x.V(\F1 F2.E(x,F2(x))))
base: manage

Here, there needs to be a declarative sentence as well as an infinitive to trigger this template. Moreover, these semantic templates can be modified and even completely rewritten, thus allowing for improvement.

The Coq file, which defines axioms, parameters, and tactics, implements meaning representations produced using the YAML file. This script calls upon different tactics to resolve a set of premises into a conclusion. The system's deductive accuracy was tested on the FraCaS textual inference problem set developed by Stanford's FraCaS consortium (Cooper et al., 1996). In FraCaS, each of the 346 problems featured comes with a set of premises, a question, and a conclusion nearly identical to the question in declarative form. The answer to each question is provided as yes, no, or unknown. The ccg2lambda framework uses the version provided by MacCartney and Manning (MacCartney & Manning, 2007), but excluding nominal anaphora, ellipsis, and temporal reference.

The overall framework is best described in Figure 6.

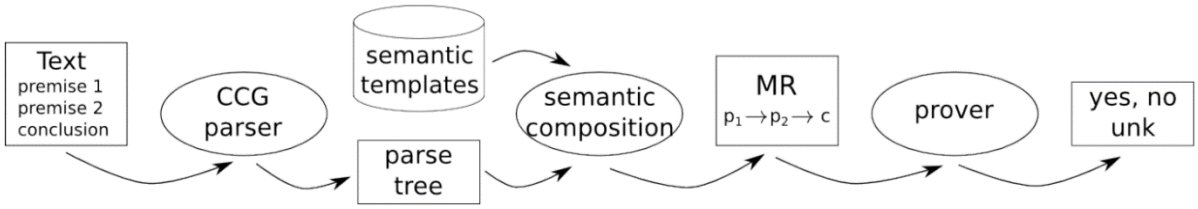

Figure 6. This is the system pipeline of ccg2lambda, directly taken from Mineshma et al., 2015.

Application

The ccg2lambda framework uploaded to GitHub by Koji Mineshima et al., no longer works due to deprecation issues. My fork has addressed these issues and has instructions for installation and execution in the Readme. It can be found at https://github.com/ajung202/ccg2lambda.

Acknowledgments

I would like to thank my advisor for the valuable insight provided to me on this topic.

References

Appel, K., & Haken, W. (1986). The four color proof suffices. *The Mathematical Intelligencer*, 8 (1), 10–20. https://doi.org/10.1007/BF03023914

Barendregt, H. P. (1984). Introduction to lambda calculus.

Barwise, J. (1977). An introduction to first-order logic. In J. Barwise (Ed.), *Handbook of mathematical logic* (pp. 5–46). Elsevier. https://doi.org/10.1016/S0049-237X(08)71097-8

Bell, J. L. (2012). Types, sets, and categories. *Sets and Extensions in the Twentieth Century*, 6, 633–687.

Bertot, Y., & Pierre, C. (2004). Interactive theorem proving and program development: *Coq'art: The calculus of inductive constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

Bezhanishvili, N., & de Jongh, D. (2006). Intuitionistic logic.

Cardone, F., & Hindley, J. R. (2006). History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 723–817.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2), 56–68. https://doi.org/10.2307/2266170

Clark, S., & Curran, J. R. (2007). Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4), 493–552. https://doi.org/10.1162/coli.2007.33.4.493

Cooper, R., Crouch, D., van Eijck, J., Fox, C., van Genabith, J., Jaspars, J., Kamp, H., Milward, D., Pinkal, M., Poesio, M., Pulman, S. (1996). Using the framework (pp. 62-051). Technical Report LRE 62-051 D-16, The FraCaS Consortium.

Coquand, T., & Huet, G. (1988). The calculus of Constructions. *Information and Computation*, 76(2-3). https://doi.org/10.1016/0890-5401(88)90005-3

Cremers, A., & Ginsburg, S. (1975). Context-free grammar forms. *Journal of Computer and System Sciences*, 11 (1), 86–117. https://doi.org/10.1016/S0022-0000(75)80051-1

Enderton, H. B. (2013). A mathematical introduction to logic. Academic.

Gonthier, G. et al. (2008). Formal proof–the four-color theorem. *Notices of the AMS*, 55 (11), 1382–1393.

Haaparanta, L. (2009). The development of modern logic. Oxford University Press.

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.

Leivant, D. (1994). Higher order logic. *Handbook of Logic in Artificial Intelligence and Logic Programming* (2), 229–322.

MacCartney, B., & Manning, C. D. (2007). Natural logic for textual inference. *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing - RTE '07.* https://doi.org/10.3115/1654536.1654575

Mineshima, K., Martínez-Gómez, P., Miyao, Y., & Bekki, D. (2015, September). Higher-order logical inference with compositional semantics. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (pp. 2055-2061).

Moore, G. H. (1988). The emergence of first-order logic. *History and philosophy of modern mathematics*, 11, 95–135.

Nozick, R. (1996). The emergence of logical empiricism: From 1900 to the vienna circle (Vol. 1). Taylor & Francis.

Steedman, M. (1996). A very short introduction to ccg. Unpublished paper. http://www.coqsci.ed.ac.uk/steedman/paper.html.

Steedman, M., & Baldridge, J. (2011). Combinatory categorial grammar. *NonTransformational Syntax: Formal and Explicit Models of Grammar*. Wiley-Blackwell, 181–224.