Sepia: a Framework for Natural Language Semantics

Gregory Adam Marton and Linda Brown Westrick

May 25, 2009

Abstract

Natural Language Understanding is the process of building machine-processable representation from segments of recorded human language. Information Extraction is one subfield of natural language understanding in which segments of a text are labelled with the kind of information they represent, for example, a news story about company mergers and acquisitions might be processed to identify the names of the companies in question, their officers, the date, the trade types and amounts involved, and other key data.

The tools described herein aim to extend that definition of information extraction to building more complex data structures from given textual inputs, about which a program might reason. For example, rather than simply identifying names, looking for persons; rather than simply marking the dates, parsing them well enough to put them on a timeline; rather than simply marking monetary amounts, being able to compare them across currencies and times. Specifically, we provide a language in which to express complex transformations from textual form to data structure, and related tools.

This report describes Sepia, a framework encompassing:

- a computer-readable representation of combinatory categorial grammar (CCG)
- a computer-evaluatable semantics for each CCG lexical item
- a parser which finds the best interpretations of CCG lexical items for a given input.
- tools for probabilistic learning for such lexical items.

The report explains the major design decisions, and is meant to teach the reader how to understand Sepia semantics and how to generate new lexical items for a new language understanding task.

1 Motivation And Overview

We implemented Sepia in order to make it easy to implement small semantic theories for restricted domains. Applications that we and several colleagues have had fun with include:[1]

- understanding dates and times and other measures: "next Wednesday"
- finding and linking names of people, organizations, places
- telling a robot what to do with things on a table: "touch the red one"
- retrieving spatial paths in video: "show people entering the kitchen"
- a toy gossip world: "John loves Mary" "Who does not hate Mary?"
- a little number theory: "18 is twice the sum of its digits."

Two of these—for names and measures—are incorporated into the START natural language question answering system[2] and help analyze queries from thousands of users a day. The purpose of these applications is to go beyond analyzing the surface structure of the inputs, beyond labelling the arguments of verbs, to actually evaluating what is meant. The date and time application knows that "a month ago" is sometime within "last month", and can identify bounds for these based in the operating system's representation; there is a physical robot that has actually performed the actions requested; the number theory program verifies (or falsifies) the statements given it, when it can. This report describes the underlying technologies behind these lexicons, and serves as a guide on how to construct them.

In building these applications, we meet, out of necessity, problems that semanticists have struggled with, including anaphoric and definite reference (the meanings of "John" and "he" and "himself"), determiners and quantifiers (the meanings of "the" and "every"), modifiers ("not", "might have"), plurals, and so on. Following Pulman [2007], we believe that systems like Sepia can help further the goals of linguistic semantics through implementation.

Manually constructed semantic applications are inherently brittle, and manual lexicon construction gets more cumbersome as each application grows. Thus one goal is to acquire new knowledge within a known domain with little human input, or with input from non-programmers. We have made first attempts at this task which formulate the problem as a search through the space of Scheme programs, but these efforts are outside the scope of this report.

The organization of the report is as follows: In section 2 we review history and related work in support of the ideas behind this view of semantics, and the major decisions that the Sepia framework embodies. In section 3 we give details of the language of semantics that Sepia provides, and the accompanying parsing implementation. In section 4, we describe an on-line log-linear model that discriminates between ambiguous possibilities. Section 5 outlines immediate opportunities, and highlights our contributions.

2 Related Work

In order to experiment with the widest variety of semantic theories, we needed a flexible lexical semantics framework. We chose a *categorial grammar* formalism (as opposed to, e.g., a context-free grammar), we

[1] Thanks to Stefanie Tellex for robots and spatial paths, to Alexey Radul for the number theory example.

[2] http://start.csail.mit.edu

chose to allow any Scheme program for the semantics of a category (as opposed to, e.g., attribute–value lists or semantic role frames), and we chose to evaluate that Scheme program at each step of the parse (as opposed to evaluating only the semantics of the syntactially best parse). We use a *log-linear model* to rank candidate parses, because such models can make use of local and global parse features equally easily. Taken together, these decisions make our parser the most expressive parsing framework as-yet implemented for natural language processing. This section puts these ideas in context, and explains and supports these decisions.

In human language we can express many more meanings through the compositional combination of words than there are individual words. The problem of how that combination happens is traditionally separated into *syntax* and *semantics* where syntax governs the rules by which words can fit together into larger meaningful units, and semantics is the study of what the combinations of those words must mean when they do fit together. These have been studied from a linguistic viewpoint, aiming to understand the human language capacity through observation, and from an artificial intelligence viewpoint, aiming to understand it through replication. There has been comparatively less work in semantics than syntax, and less in joining the two viewpoints.

From the point of view of syntax, while context-free grammars, due to their simplicity, have been the choice of many well-known natural language parsers, they are not expressive enough to account for all natural language phenomena [Kac et al., 1987, Manaster-Ramer, 1987][3]. Many newer parsers therefore support a grammar from a more expressive family of grammars called mildly context-sensitive grammars (MCSGs). For Sepia we have chosen Mark Steedman's combinatory categorial grammar (CCG), which is a mildly context-sensitive grammar. The central difference is that mildly context-sensitive grammars allow one to specify parse subtrees in a parse tree, rather than a single level of nonterminal, so it is possible to enforce grammatical agreement between arbitrarily long subphrases. Despite their additional power, CCG parsers have been implemented efficiently enough for wide-coverage use [Doran and Srinivas, 1997, Clark and Curran, 2007].

We chose combinatory categorial grammar (CCG) in particular because of its ties to lambda calculus, the theoretical basis for functional languages like Scheme. In CCG, the meanings of words are shepherded along the parse tree for an input, combining into meanings of longer phrases and eventually into a meaning for the entire input. Each of these meanings is specified, in our implementation, as a Scheme program, and the Scheme interpreter evaluates meanings as they get combined. Of course, not all Scheme programs will combine without error, which adds to the expressivity of our language: parses whose semantics don't work out can be abandoned early.

Blackburn and Bos [2003] introduce the "relatively new discipline" of computational semantics, and advocate for a first-order logic approach. They acknowledge that "Traditional formal semantic analyses of human language typically presuppose formalisms with high-expressive power", but suggest that for a computational semanticist, restricting oneself to first-order logic offers the benefits of existing efficient automatic theorem provers to make inference possible, and yet the representation is "able to deal (at least to a good approximation) with a wide range of interesting phenomena". Indeed several existing natural language processing frameworks, those that incorporate semantics at all, use first-order logic [Zelle and Mooney, 1993, Loper and Bird, 2008, Zettlemoyer and Collins, 2007, Bos et al., 2004], or the slightly more expressive hybrid logic [Baldridge and Kruijff, 2002].

Despite these systems being available, Pulman [2007] laments "formal semanticists hardly ever give

[3]but note that the phenomena that Kac et al. [1987], Manaster-Ramer [1987] rely on might be explainable using a language with expressive power between that of CFG and MCSG [Savitch, 1989]. Nonetheless, most of the recent work has been done with MCSG.

	Grammar	**Semantics**
He & Young 2005	Context-free	Attribute-value lists
Yi & Palmer 2005	Context-free	Semantic role labels
Wong & Mooney 2005	Context-free	Prolog
Baldridge 2002	CCG	Hybrid logic
Clark & Curran	CCG	first-order logic
Zettlemoyer & Collins 2007	CCG	Lambda calculus in Java
Sepia	CCG	Scheme

Table 1: **Some existing semantic parsers**: Sepia is not the only one to use the convenience of an existing programming language for semantics, but it is the first to combine such executable semantics with a mildly context-sensitive grammar formalism. It is also the first to evaluate its semantics *at the same time* as its syntax, and thus allow the semantics to influence the syntactic parse.

any thought to how their analyses could be mechanised, either for the purposes of testing the analysis to ensure that it does what it is supposed to do (and does not do what it is not supposed to do), or how it could be used in the context of some practical application like database query or question answering." Pulman attempts to bridge this gap by implementing Klein's theory of comparatives in first-order logic. Klein's formulation uses a higher-order logic, but out of the same expedienceas Blackburn and Bos, first-order seems like the way to reach practical applications. He does this while going further than Blackburn in declaring first-order logic's inadequacy: "many natural language constructs are known not to have a natural formalisation (perhaps not any formalisation at all) in first-order logic and so we seem to be stuck with an ineradicable higher order component". He is successful largely successful, but describes limitations to the approach as well, in which e.g. explaining the use of more than one gradeable adjective, like "an expensive cheap hotel", "is probably impossible in principle, and not very compositional even if it were possible". And such attempts are rare: "I do not know of any other implemented analysis that manages to capture as many of the relevant inferences as this one".

By contrast, as early as the 1970s, in artificial intelligence, very powerful semantics had made an appearance in Winograd [1971] in the form of procedural knowledge representation, and later in Bobrow and Winograd [1977] and Winograd [1980] as "meaning as command", incorporating implicature and calling for development of reasoning far beyond deductive logic. What these early systems lacked was a formulation for learning semantics or even lexical preferences from corpora. Recently, Zelle and Mooney [1993] and Zettlemoyer and Collins [2007] showed that learning was possible for first-order semantics if first-order semantic ground truth expressions were given as supervision. Because Sepia works instead in the space of programs, these results only translate insofaras a Sepia lexicon happens to be implementing first-order logic. The challenge that Sepia lets us formulate is whether it is possible to learn other kinds of semantics, whether higher-order formal semantics or procedures, in a similar framework.

As a final point of motivation, there have been two sets of shared evaluations recently where compositional computational semantics has been put to the test: the PASCAL Recognizing Textual Entailment Challenge (RTE) and the Time Expression Recognition and Normalization task (TERN). These are in fact both multi-year series of evaluations that have evolved, but the core tasks remain true to the ideas at inception. The RTE task is, given a short basis text and a hypothesis, to decide whether the hypothesis can be inferred from the basis text, as that would typically be interpreted by people. For example, from "Haifa University in Israel was founded in 1978", the system must decide that the hypothesis "Israel was

founded in 1978" does *not* follow. The TERN task involves deciding what date an expression like "last Wednesday" refers to in news articles (and more recently transcripts and texts of other genres), as well as identifying durations, periodicities, and other time expression phenomena. In every case, more than half of the participants manually created rule-based resources of some kind, generally alongside some machine learning [Ferro, 2004, NIST, 2006, 2007, Dagan et al., 2006, Bar-Haim et al., 2006, Giampiccolo et al., 2007]. A common framework for such rules, flexible enough to accomodate them, and easy to integrate with machine learning approaches, seems a worthwhile goal.

Our CCG+Scheme system is the most flexible semantic parsing combination built to date, yet with this flexibility is fast enough for practical applications. One cannot theoretically analyze Sepia's syntax as mildly-context sensitive, however, but must conclude that even the syntax is Turing-complete, because of a decision we have already introduced: evaluating semantics at each step. In this formulation, the semantics can cause the syntax to abandon a parse early, and so influence the set of utterances that are grammatical in the strict sense. In section 3.4.6, we describe a mechanism that goes further in embracing the Turing-completeness of the syntax, and asks us to view CCG as a convenient heuristic, rather than a hard constraint. It is up to the (perhaps automatic) lexicon implementer not to abuse this ultimate flexibility.

3 The Mechanics of the Parser

This section presents a unified overview of Steedman's CCG syntax and our form of Scheme semantics, intentionally mixing ideas from the two, and from other sources, for clarity and simplicity of presentation. It is intended for those who wish to begin using our system, and want an understanding of underlying concepts without separately reading the prior work. That said, we highly recommend reading (at least) the first three chapters of The Syntactic Process [Steedman, 2000] for the rationale behind, and advantages of, CCG. For those already familiar with CCG, our novel contributions are listed in section 3.7, with links to the more detailed explanations within the tutorial.

The report assumes a working knowledge of Scheme. We have implemented both the software in general and the language of semantics in particular in the GNU/Guile[4] variant of Scheme, so such knowledge will be necessary for any user. If the reader is familiar with programming but unfamiliar with Scheme, it may be enough for the exposition to know that expressions are surrounded by parentheses, and that the first item in an expression is generally the name of a function, and the rest of the items are its arguments. One introduces a new function using the keyword "lambda", followed by the argument list to the function, and finally the body of the function—what it should do with the arguments. This is a tremendous oversimplification of the language, and for a more useful introduction, we recommend Sitaram [2004].

As a matter of Scheme notation, we will abbreviate the word "lambda", use dot-and-comma notation for the arguments, and use other mathematical notation, so that where in Scheme one would write `(lambda (arg1 arg2 ... argN) body)` we will write $(\lambda.\texttt{arg}_1,\texttt{arg}_2,\ldots,\texttt{arg}_N\ \texttt{body})$, or in the usual one-argument case, $(\lambda.\texttt{arg body})$.

3.1 Lexical Items as Units of Meaning

In a Combinatory Categorial Grammar (CCG) lexicon, each word or phrase is associated with one or more *categories*, with each category corresponding to one fine-grained meaning for the word. We will call the

[4] http://www.gnu.org/software/guile/

word or phrase a *pattern* to indicate that we will look for it in the input as a first step in processing. A pattern and a category together, so a word and a single fine-grained meaning, are called a *lexical item*.

A category has two parts, a *signature* and a *semantics*. The signature serves to restrict which categories can combine. The semantics specifies what should happen when they do combine. The goal of the parser is to enumerate the possible combinations of categories and to evaluate their semantics.

We will present this first overview based on the example in Figure 1, in which we will parse "three twenty three", whose final meaning should be the number `323`. This was among the motivating examples in our initial search for a semantic parser, and is the sort of semantics that is either difficult or less elegant to express in other languages of semantics.

We restrict the first example to an extremely simple vocabulary in which the only meanings are those for the words "three" and "twenty", and each of these has just two meanings. Recall that a lexical item is a pattern associated with a category, and that a category itself has a signature and a semantics, so each lexical item will have three parts. We will write them as

(1) pattern := signature : `semantics`

The distinguishing factors in this notation include color and punctuation. The pattern is presented in purple, followed by a colon and equal sign, then the category. The category has the signature first, in brown, then a colon and the semantics in green. The punctuation is standard in the literature, and the colors are used for emphasis throughout this report. In Scheme, a lexical item is written (ccg/lexical-item pattern signature `semantics`) which, because it is code, omits the punctuation. We use the notation in (1) for the exposition because it is easier to read, but it is useful to remember that every example can be trivially translated into a lexical item readable by Sepia.

Our four example lexical items for "twenty" and "three" are:

(2) "three" := ones : `3`
"twenty" := tens : `20`
"twenty" := tens/ones : `(λ.ones (+ 20 ones))`
"three" := hundreds/tens : `(λ.tens (+ 300 tens))`

The first two are intuitive—"three" means 3, and is a ones-place type of number, "twenty" means 20. These two cannot combine with each other. The slashes in the next two signatures signify that they can combine with other signatures, which other signature they expect, and in what direction. A signature with a slash in it is in some sense incomplete—we call such signatures *unsaturated*, whereas the signatures of the first two lexical items were *saturated*.

The unsaturated meaning for "twenty" expects a ones to combine with; it expects this to its right, as signified by the forward (/) rather than backward (\) slash; and upon successfully combining, it will produce a tens. These syntactic rules for combining are formally specified and refined in Section 3.4. Likewise, the unsaturated meaning for "three" expects a tens on its right, and will produce a hundreds. These combinations can happen in the abstract, but do not happen in fact until we get an input string and begin processing it.

3.2 Processing Input

When a pattern is seen in an input sentence, the first step of processing is to *recognize* it. The example in Figure 1 shows an input string, the patterns that match from the input, and their associated categories **a**

through **f**. Recognition is the process of finding patterns that match the input, and putting their categories into a data structure called the *parse chart*. Recognition is described in greater detail in Section 3.3. Some notes on details of chart parsing relevant to Sepia are given in Section 3.6.

A parse chart is an efficient representation for the possible ways that items might combine. Each set of possible combinations is called a *derivation* or a *parse tree*. The entire set of possible parse trees, hence all the possibilities represented in the chart taken together, are called a *parse forest*. The central problem in Section 4 is deciding which of the possible parse trees is the best for a given input string.

A parse tree is a metaphor for a set of combination choices. In the metaphor the leaf locations are at the top, near the input, and span just one input token (**a**–**f**), while the root node is at the bottom, and spans the greatest number of input tokens (**i**). The number of input tokens that a category spans is called its *level* in the parse tree. In much of the linguistic literature, parse trees are written with the root at the top and the leaves at the bottom, but we follow the standard notation for CCG, as well as the intuition of the metaphor, in maintaining this order.

Parsing—the process of finding the possible sets of combinations – proceeds by considering each adjacent pair of categories and combining them into any new categories that would span both. The category **b** can combine immediately with **c** to form **g** because it requires a tens to its right, as indicated by the forward slash, and **c** has a signature tens, and is immediately to **b**'s right. In the next round, once **g** and **h** have been created, **b** can combine with **h** to form **i**. Each category represents an interpretation of the phrase that it spans, so category **g** is a meaning for "three twenty" out of context, and **i** is a meaning for the entire input, "three twenty three".

A derivation (a.k.a. parse tree) is, in this sense just one path through the parse chart. While every category corresponds to a parse, and every saturated category corresponds to an easily interpretable parse, we will often write the derivation for just the one or two parses that we're interested in.

We show the derivation of `323` at the bottom of the figure. This derivation corresponds to a parse tree rooted at **i** with children **b** and **h**, where **h** has children **d** and **e**. The first line of the derivation reflects the input. Each subsequent line of the derivation corresponds to a level of the parse tree, and the arrows at the end of each separator indicate the direction of application. Notice that patterns are only used for lexical recognition, and that only the categories get carried around thereafter.

Semantics in CCG closely follows syntax. The semantic content associated with each word or phrase is expressible as a lambda expression, or for us, a Scheme program. Whenever two categories combine, the one with the slash must be a function. It receives as its argument the semantics of the adjacent category with which it combined. The return value of this function is used as the semantic value of the combined expression. In the example, we applied the function `(λ.ones (+ 20 ones))` to the argument `3` to produce `23` (via the intermediate expression `(+ 20 3)`). We then applied the function `(λ.tens (+ 300 tens))` to `23` to produce `323`.

We can now go into greater detail.

3.3 Lexical Recognition

Given an input string, we break it into tokens at each possible word boundary. We look for lexical items whose string patterns match each token. Every lexical item has an associated category, namely a signature–semantics pair, and there may be multiple lexical items with matching patterns. A parse chart is created for the input, with the tokens as leaves, and we insert all of the categories found into the positions in the chart corresponding to the words they came from.

Compositionality is a base assumption of semantic theories—somehow the meanings we know for in-

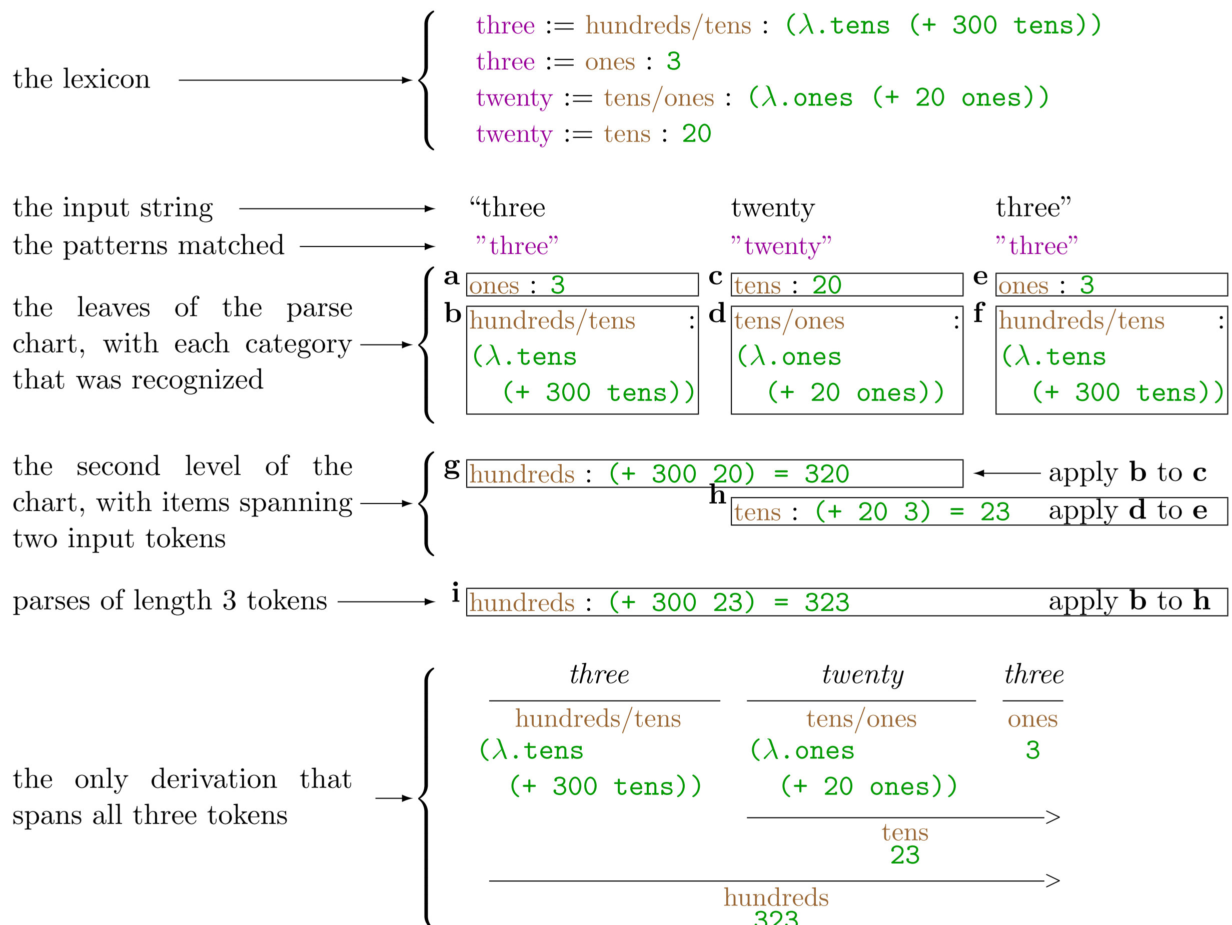


Figure 1: A simple numerical parse example: at top are a lexicon and an input string. We see the patterns that were recognized, and the parse chart of possible categories. Categories are the signature : semantics pairs shown in boxes and labelled **a**–**i**. The derivation for the largest parse is shown at bottom.

dividual items combine to form many more possible meanings than words. A lexical item whose pattern is the string "new", might have an adjective-like signature, looking for a noun to modify, and a semantics that restricts the noun to somehow be new. Similarly, "jersey" may have a noun signature and a semantics that specifies something about outerwear covering the torso and arms. By contrast, we may wish to have a separate meaning for the phrase, "New Jersey", not only because its elements are usually capitalized, but also because it may refer to something altogether different than the combination of the meanings of "new" and "jersey".[5] Sepia allows lexical items with multi-token patterns, like the string "New Jersey", and puts their associated categories into the parse chart in the usual way. In such cases, both the compositional and the non-compositional readings will be available for that phrase in the parse.

Each lexical item may choose to be individually case-sensitive or not, and it is possible to use the entire lexicon case-insensitively for, e.g., broadcast news transcripts, which do not have case information.

In addition to string patterns that must match exactly, Sepia supports regular expression patterns. When a regular expression matches an input token in full[6], then that lexical item is recognized.

[5] albeit perhaps retaining some compositional semantics of the same "new" and a different "Jersey" in England

[6] e.g., the regular expression pattern /a.*/ will match the token "ab" but not "lab", because there are implicit anchors

Recognition is more complex for regular expression patterns than for string patterns, because the semantics is not simply inserted into the chart verbatim. When a regular expression matches, the parser generates a match structure which it expects to pass to the semantics as its only argument. The semantics for the lexical item must therefore be one lambda deeper than usual: the outermost lambda takes this match structure and returns the actual semantics to be used. For example, for the regular expression pattern /(\d+)/, which matches tokens consisting entirely of digits, the corresponding semantics in the lexical item may be `(λ (match-struct) (string->number (match:substring match-struct 1)))`. This takes the portion of the token matching the first parenthesized subexpression (in this case the entire digit string), and applies the Scheme function `string->number`, so that the result, a number, will be the initial semantics corresponding to this lexical item in the chart.

It is possible to use regular expression patterns on the entire input string rather than on each token. This is convenient for identifying URLs[7] or significant whitespace, for example, but is otherwise generally eschewed in favor of compositional ways to understand an input phrase.

Once the input string has been broken into tokens and a parse chart has been populated with the categories from recognized lexical items, we are ready to start combining these categories into larger parses.

3.4 Combinators

In order to combine categories into larger parses, one must define the rules by which they can combine. In CCG these rules are called *combinators*. When deciding how a semantic framework will work, a theory can trade off complexity in the lexical items for complexity in the combinators. CCG has at its core two simple but powerful combinators: *functional application* and *type change*, and a number of other combinators that can be seen as extending these or using them in a special way. In effect, these are the simplest combinators one can imagine, putting most of the burden of specifying meanings on the lexical items themselves.

3.4.1 Functional Application

We have spoken about signatures so far only in the abstract, noting that signatures specify how a category will combine with others. A signature specifies, in particular, the set of other signatures that a category expects to combine with, the direction (left or right of itself in the input string) that it expects to find the other categories with those signatures, and the signature that results from these combinations.

A signature that specifies a simple type, X, represents a value that does not expect to combine with anything. A signature X/Y expects to find an item with signature Y, and produce from it a new item with signature X. The forward slash indicates that Y is expected to the right of the item with signature X/Y, whereas a backslash indicates that the desired category is on the left. Thus for functional application:

(3) $\mathrm{X/Y} : \mathtt{f} \quad \mathrm{Y} : \mathtt{a} \;\Rightarrow\; \mathrm{X} : \mathtt{(f\ a)}$ (“forward functional application” or >)
$\mathrm{Y} : \mathtt{a} \quad \mathrm{X\backslash Y} : \mathtt{f} \;\Rightarrow\; \mathrm{X} : \mathtt{(f\ a)}$ (“backward functional application” or <)

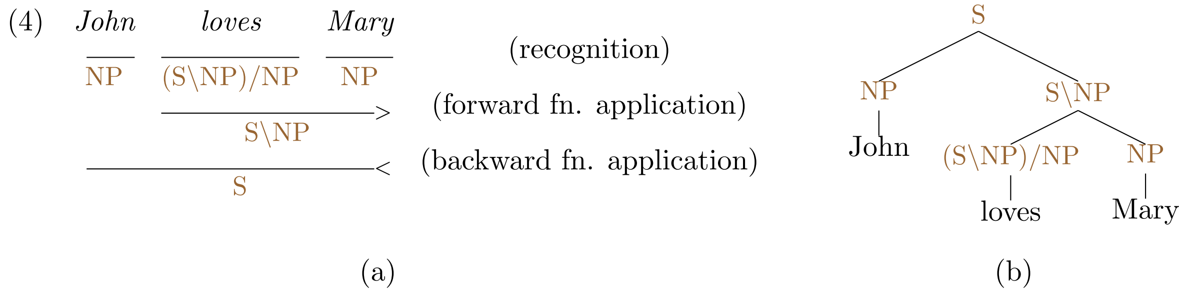
The semantics get carried along: whenever a signature has a slash, that indicates that it is a function that can be applied per the rules above, so its scheme semantics must be a function of one argument.

that turn it into /^a.*$/

[7]A URL is clearly made up of parts – a scheme, hostname, directory, query, and parts within those, and it would be possible to build up a URL from its individual tokens, its individual sequences of numbers, letters, and punctuation. A regular expression pattern can help delineate the entire url, and treat it as a single token, without having to write a lexicon that understands URLs via their pieces.

When the combination is made, the Scheme function is immediately applied to its argument.

If a signature has more than one slash, then the function must be *curried*—each semantics can be a function of only one argument, but instead of a value result, it can return another function of one argument. A standard example for such a function is a transitive verb, e.g., “loves”. Leaving aside semantics for a moment, consider a two-step derivation for “John loves Mary”:

(4) 

(a) (b)

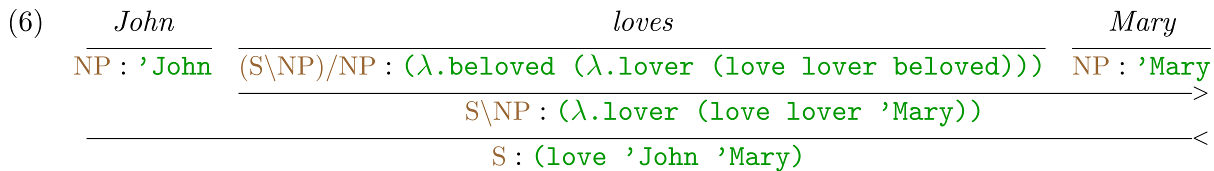
Examples 4a and 4b are meant to express the same derivation; those who are more used to looking at syntax trees in linguistics may find 4b clearer, but we will use the format of 4a, in part because it also marks the CCG rule by which each line was derived.

Note that we have added parentheses in the signature for loves to indicate association. By default, CCG is leftward associative, so these parentheses in particular are redundant, but parentheses are essential (and supported in Sepia) when seeking to combine with an unsaturated category, as we will see a few examples hence.

We have here assumed three lexical entries, for which we will now also specify some semantics:

(5) “John” := NP : `'John`
“Mary” := NP : `'Mary`
“loves” := (S\NP)/NP : `(λ.beloved (λ.lover (love lover beloved)))`

At the recognition stage of the derivation, we recognize the words in the input string based on the patterns in the lexicon. All three lexical items use simple string patterns which match a word in the input string exactly, and three new categories are made, each of which contains the signature and semantics from the lexical items that were recognized.

(6) 

The category for “loves” now applies rightward to the category for Mary, producing a new category with signature S\NP, with a semantics `(λ.lover (love lover 'Mary))`. The outer function has been applied to the semantics `'Mary`, returning a function that expects a lover, and has already filled in Mary as the beloved. In the final step, this new category applies leftward to the category associated with “John”, producing finally the category with signature S and semantics `(love 'John 'Mary)`.

Note that we use a function, `love`, that is not defined by Scheme. By contrast to other work, in our implementation this would result in an error (unbound variable), and the parse would be abandoned. For

the purpose of this exposition, the semantics of `love` may as well be the same as `list`, but we're sweeping vast amounts of meaning under a very thin rug. If we want to represent frame semantics or grammatical dependencies or lambda calculus formulae, we must create those data structures explicitly using valid Scheme programs for each word.

Question syntax and relative clauses in CCG are not handled through movement, but by applying to an unsaturated category. For example for "Who loves Mary":

(7)

Who	*loves*	*Mary*
Q / (S\NP)	(S\NP)/NP	NP
	S\NP (>)	
Q (>)		

The semantics for "Who" depends a great deal on what kind of data structures one is expected to search. As an example, suppose that we have a list of subject–relation–object triples, and a function `relation-matches?` that treats the symbol `'any` in a special way. Let us then use the following sketch of a semantics[8]:

(8)

```
"Who" := Q/(S\NP) : (λ.make-query-reln
                      (λ.context
                        (filter
                          (λ.kb-reln
                            (relation-matches? (make-query-reln 'any) kb-reln))
                            ;; relation-matches returns whether a query relation matches
                            ;; any relation in the knowledge base of relations provided
                          context)))
"loves" := S\NP/NP : (λ.y (λ.x (make-relation "love" x y)))
"John" := NP : 'John
```

When the final semantics are computed, in this example, the function would be applied to a knowledge base or context, which here is just a list of relations. The result is the set of matching relations, and the outer program can form an appropriate response based on those. This example is intended to give a flavor of how one might construct a useful system that takes advantage of the CCG syntax and executable semantics that Sepia provides.

3.4.2 Type Change

In many cases we want a constituent to take on multiple roles with related semantics. As a simple example, "02139" may simply be a number somewhat larger than two thousand, but it may have another interpretation as an identifier (e.g., a serial number) or perhaps more specifically a ZIP code[9] (for part of Cambridge, MA, USA). Likewise, though we sometimes want to treat spelled-out numbers differently from strings of digits (zipcodes, for example, are not generally spelled out in written text), for other purposes we

[8]It is a sketch because we neglect to define most of the details of the helper functions and the context, omitted largely because their details are both difficult and not relevant to the core of the exposition.

[9]five-digit "Zoning Improvement Plan" codes for administrative districts in the United States, hereafter called "zipcodes" to avoid unnecessary emphasis

might like to treat both as numerical values. CCG provides a facility called a unary type-change (different from type-raising) that makes these multiple interpretations possible.

Type-change operates on just one category in a parse, rather than combining two of them, and can change the resulting syntax and semantics arbitrarily.

(9) X : `x` ⇒ Y : `(f x)` ("type change" or ↑)

Sepia allows one to specify a type change by using a signature, rather than a string or regular expression, as the pattern of a lexical item. When the pattern is a signature, the item will match any category during parsing which has that signature, and its semantics will apply to the semantics of that category to create the semantics of the new category.

For the "02139" example, a regular-expression pattern may recognize it as a digits-string, and then it may be type-changed to zipcode, using the following lexicon:

(10) /\d+/ := digits-string : `(λ.match-struct (match:substring match-struct 0))`
digits-string := zipcode : `(λ.d (if (equal? 5 (string-length d)) d (fail-this-parse!)))`
"twenty" := spelled-out-number : `20`
digits-string := number : `(λ.x (string->number x))`
spelled-out-number := number : `(λ.x x)`

The derivation is entirely vertical:

(11)

(a)	(b)	(c)
02139	*213*	*twenty*
digits-string : `"02139"`	digits-string : `"213"`	spelled-out-number : `20`
↑	↑	↑
number : `2139`, zipcode : `"02139"`	number : `213`	number : `20`

Note that type-change also allows us to create infinite loops, changing a category to itself repeatedly and never continuing with the rest of parsing. Lexicon developers (both human and automatic) must take care to keep the directed type-change graph acyclic.

3.4.3 Composition and Type Raising

Returning to John and Mary from Section 3.4.1, let us examine the mirror question: "Who does John love?" It is very similar, but using the same signatures doesn't work, first because we haven't specified the lexical item for "does" (or untensed "love"), but more importantly because the signature for "Who" that we used for "Who loves Mary" expected a signature S\NP on its right, and with "John love" we will have something of the form S/NP, expecting an argument to its right.

Ignoring tense for the moment for clarity, we'll assume the word "does" is meaningless, and that other new items are similar to those already described:

(12) "love" := S\NP/NP : `(λ.y (λ.x (make-relation ''love'' x y)))`
"does" := S/S : `(λ.x x)`

```
"Who" := Q/(S/NP) : (λ.make-query-reln
                       (λ.context
                         (filter
                           (λ.kb-reln
                             (relation-matches? (make-query-reln 'any) kb-reln))
                           context))) ;; same semantics as before
```

But "love" can no longer apply right as the first step. We use the CCG combinators *type raising* and *composition* to allow it to effectively apply to the subject first, to generate S / NP, from where the derivation is straightforward.

(13)

```
 Who        does   John        love
--------    ----   -----    ----------
Q/(S/NP)    S/S    NP       (S\NP)/NP
                   ----->T
                 S/(S\NP)
                 --------------------->B
                          S/NP
            -------------------------->B
                       S/NP
------------------------------------------>
                  Q
```

Type raising (>**T**) turns an argument of some function f into a new function whose argument is f. In the derivation above, it turns "John" from an argument into a function over something (like "love") that wants its own type, NP to its left. It remembers (or in programming language terms "closes over") the original argument.

Composition (>**B**) allows us to package up two adjacent functions, where some product of one function (the composee) is the same type as the outermost argument to the other function (the composer) and the composer wants to see that product on the appropriate side. The type-raised version of "John", whose signature is S/(S\NP), is the composer, looking for a product S\NP to its right. The possible products of S\NP/NP are S\NP with one functional application or S with two. The first of those, S\NP, does match what the composer is looking for, so the composition can proceed. Once these have been packed into a single S/NP, we compose again, this time with S/S looking for the product S on its right. Finally, we apply "Who". The syntactic and semantic rules for type raising and composition are as follows:

(14)

X : x		$\Rightarrow_{\mathbf{T}}$ Y/(Y\X) : (λ.f (f x))	("forward type raising" or >**T**)
X : x		$\Rightarrow_{\mathbf{T}}$ Y\(Y/X) : (λ.f (f x))	("backward type raising" or <**T**)
X/Y : f	Y/Z : g	$\Rightarrow_{\mathbf{B}}$ X/Z : (λ.x (f (g x)))	("forward composition" or >**B**)
Y\Z : g	X\Y : f	$\Rightarrow_{\mathbf{B}}$ X\Z : (λ.x (f (g x)))	("backward composition" or <**B**)

For completeness, we will also follow the semantics through:

(15) *Who* *does* *John* *love*

```
Who                  does         John                    love
-----------          -------      ----------------        --------------------------------------------------
Q / (S\NP)           S/S          NP                      (S\NP)/NP
(λ.make-...)         (λ.x x)      'John                   (λ.y (λ.x (make-relation ''love'' x y)))
                                  ----------------->T
                                  (λ.f (f 'John))
                                  ---------------------------------------------------------------------------->B
                                    (λ.c1 ((λ.f (f 'John))
                                          ((λ.y (λ.x (make-relation ''love'' x y))) c1)))
                                                   which reduces in lambda calculus to
                                  (λ.c1 ((λ.f (f 'John)) (λ.x (make-relation ''love'' x c1))))
                                                   which reduces in lambda calculus to
                                      (λ.c1 ((λ.x (make-relation ''love'' x c1)) 'John))
                                                   which reduces in lambda calculus to
                                          (λ.c1 (make-relation ''love'' 'John c1))
                     ----------------------------------------------------------------------------------------->B
                        (λ.c1' ((λ.x x) ((λ.c1 (make-relation ''love'' 'John c1)) c1')))
                                               which reduces in lambda calculus to
                                 (λ.c1' ((λ.x x) (make-relation ''love'' 'John c1')))
                                               which reduces in lambda calculus to
                                     (λ.c1' (make-relation ''love'' 'John c1'))
---------------------------------------------------------------------------------------------------------------->
                                     an expression that finally reduces to
     (λ.context
       (filter
         (λ.kb-reln
           (relation-matches? (make-relation "love" 'John 'any) kb-reln))
         context))
```

Composition can introduce spurious parses, which have the same semantics as what we would think of as the default parse, but arrived at it in a different order. For example, if adjectives have a category n / n (something expecting a noun to its right and yielding a (now modified) noun), then “big blue tent” could be parsed by first grouping “blue” with “tent” and then grouping “big” with those (the default) or by first grouping “big” with “blue” through composition, and then applying to “tent”. This can be useful in analyzing coordinations like “big blue and little green tents” because once “big blue” and “little green” become constituents through composition, it is easy for the semantics of “and” to recognize that they are, in some sense, the same, and so to combine them (more on that soon). At the same time, it means that parsing naively, we would have multiple equivalent copies of the same parse. Likewise, in the type-raising example above, once “John” has raised to S / (S\NP), it could compose first with “does” S / S, and the question would have another parse with the same semantics. In order to prevent these spurious ambiguities, we implement Eisner Normal-Form parsing [Eisner, 1996], and extend it to prune type-raising in a similar fashion.

Some readers will have noticed that a type-raise is like a specialized dynamic type-change: one type changes to another in a predictable pattern based on the type next to it, and the type-change semantics are always the same: `(λ.x (λ.f (f x)))`. Recall that type-change semantics transforms the semantics associated with the starting type into the semantics associated with the result type, so `x` there is the semantics of the starting type, the np in our question, and `f` is the verb that will take it as subject. This

semantics remembers `x` and applies `f` to it.

In describing the type-raising operator, Steedman [2000] says that particular languages should be able to restrict it to certain categories, and indeed, implementing it in the general case is computationally expensive. Sepia asks lexicon creators to allow individual type-raises by specifying them as type changes with the special semantics `*ccg/type-raise*` (defined as described above). By using the predefined semantics for the type-change, the lexicographer lets the parser treat these specially, pruning spurious semantics as one would for composition, and applying only as needed.

3.4.4 Cross-Composition and Generalized Composition

Cross-composition is the first combinator we introduce that will let us scramble words or phrases in a sentence, rather than strictly preserving order. We give the rules for cross-composition, and then we merge composition and cross-composition (following Eisner [1996]) into a generalized composition combinator, which is also generalized in the sense of taking an arbitrary number of arguments of either direction (indicated by a vertical bar):

(16)

$$\mathrm{X/Y} : \mathtt{f} \quad \mathrm{Y\backslash Z} : \mathtt{g} \;\Rightarrow_{\mathbf{B}}\; \mathrm{X\backslash Z} : (\lambda.\mathtt{x}\ (\mathtt{f}\ (\mathtt{g}\ \mathtt{x}))) \qquad \text{(“forward cross-composition” or } >\mathbf{B}_\times)$$

$$\mathrm{Y/Z} : \mathtt{g} \quad \mathrm{X\backslash Y} : \mathtt{f} \;\Rightarrow_{\mathbf{B}}\; \mathrm{X/Z} : (\lambda.\mathtt{x}\ (\mathtt{f}\ (\mathtt{g}\ \mathtt{x}))) \qquad \text{(“backward cross-composition” or } <\mathbf{B}_\times)$$

$$\mathrm{X\backslash Y} : \mathtt{f} \quad \mathrm{Y}\ |_n \mathrm{Z}_n \ldots |_2 \mathrm{Z}_2 |_1 \mathrm{Z}_1 : \mathtt{g} \;\Rightarrow_{\mathbf{B}} \qquad \text{(“generalized forward composition” or } >\mathbf{B}n)$$
$$\mathrm{X}\ |_n \mathrm{Z}_n \ldots |_2 \mathrm{Z}_2 |_1 \mathrm{Z}_1 : (\lambda.c_1\ (\lambda.c_2 \ldots\ (\lambda.c_n\ (\mathtt{f}\ (\ldots\ ((\ldots\ (\mathtt{g}\ c_1)\ c_2))\ c_n)))))$$

$$\mathrm{Y}\ |_n \mathrm{Z}_n \ldots |_2 \mathrm{Z}_2 |_1 \mathrm{Z}_1 : \mathtt{g} \quad \mathrm{X\backslash Y} : \mathtt{f} \;\Rightarrow_{\mathbf{B}} \qquad \text{(“generalized backward composition” or } <\mathbf{B}n)$$
$$\mathrm{X}\ |_n \mathrm{Z}_n \ldots |_2 \mathrm{Z}_2 |_1 \mathrm{Z}_1 : (\lambda.c_1\ (\lambda.c_2 \ldots\ (\lambda.c_n\ (\mathtt{f}\ (\ldots\ ((\ldots\ (\mathtt{g}\ c_1)\ c_2))\ c_n)))))$$

These are important for handling, among other things, adverbs, including temporal adverbials. Consider that “tomorrow” can appear almost anywhere in a sentence:

- “Tomorrow, John will love Mary.”
- “John tomorrow will love Mary.”
- “John will tomorrow love Mary.”
- “John will love Mary tomorrow.”

Of course not all adverbs behave this way (consider “very”) but if the signature of this kind of adverb is S|S (which is to say, either S/S or S\S, easily decomposed into 2^n ambiguous lexical items when a signature containing a vertical bar is added to the lexicon), then with generalized composition the rest becomes easy. In the first and last case, the sentence parses as usual, and “tomorrow” finds a complete sentence and can simply apply. The interesting cases are when “tomorrow” is within the sentence:

(17)

John	*will*	*tomorrow*	*love*	*Mary*
NP	S/S	S|S	(S\NP)/NP	NP

- *love Mary*: (S\NP)/NP NP ⇒ S\NP ($>$)
- *tomorrow* (*as* S/S) + S\NP ⇒ S\NP ($>\mathbf{B}_\times$)
- *will* S/S + S\NP ⇒ S\NP ($>\mathbf{B}_\times$)
- *John* NP + S\NP ⇒ S ($<$)

Note that, even though we used the notation for cross-composition above, both composition and cross-composition are special cases of generalized composition, which is what Sepia implements.

Also note that, like other CCG parsers [Clark and Curran, 2007], we have not actually implemented general type raising, but allow particular common types to raise in particular ways using the type-change mechanism. Each lexicon must enumerate these type raises. Allowing generalized type raising would add a processing cost exponential in the length of the input.

3.4.5 Adjacent Functional Application

With Sepia, we introduce a new functional application operator, adjacent-application, motivated by the meaningfulness of (a lack of) whitespace. Sepia does not expect pre-tokenized input, and in fact records the original character positions of each of its tokens and phrases.[10] We found, in looking at such input, that whether or not there is whitespace between two tokens can be helpful in figuring out what they mean. For example in recognizing numbers, "123, 456" is quite different from "123,456"; in particular, the first is much more likely to be two distinct integers in a list, whereas the second is more likely to be a single integer in the hundreds of thousands range.

(18) X/-Y : `f` Y : `a` ⇒ X : `(f a)` ("forward adjacent functional application" or >−)
Y : `a` X\-Y : `f` ⇒ X : `(f a)` ("backward adjacent functional application" or <−)

Adjacent functional application is like functional application, but enforces that the tokens that it combines must have no intervening whitespace. In the general composition case, our notation for vertical bar must also include these combinators, and there will be two more cases for generalized forward adjacent composition and generalized backward adjacent composition. Type raising remains unaffected, because regular forward functional application works in the adjacent case, so it is only adjacent functional application which is defined not to happen in the non-adjacent case.

3.4.6 Coordination, Matching Pairs, and the λ-signature

One of CCG's successes is its ability to handle coordination not only in cases where the phrases being coordinated are standard constituents, but also by first finding non-standard constituents via type raising and composition, and then being able to coordinate those as well. The coordination combinator, like a type change, cannot be applied without further specification, because each instance of the coordination operator will have different semantics. Thus again in Sepia we treat it as a special kind of lexical item.

The primary function of the coordination operator (Φ) is to look for elements of the same kind on either side of itself. For specific signatures to the right and left, of course, it's easy to write a lexical item that will coordinate them. Rather than create a special lexical item format just for the general case, in Sepia we decided to allow signatures to evaluate any function at all, and provide the standard Φ functionality as a library call within that framework.

If a signature is specified as a procedure, then that procedure will be evaluated on every possible combination, and if it returns a new signature (rather than `#f`), then that will be the signature of the result of the combination. The semantics will apply as usual.

(19) fn-sig : `f` X : `x` ⇒ (fn-sig X '/) : `(f x)` (forward lambda application)
X : `x` fn-sig : `f` ⇒ (fn-sig X '\) : `(f x)` (backward lambda application)

[10] In the spirit of standoff XML, the purpose is to allow markup of an original text with overlapping elements, and without having to modify that text. Initiative [2007]

$X : \texttt{a} \quad \Phi : \texttt{f} \quad X : \texttt{coord} \Rightarrow \quad X : \texttt{((coord a) b)}$ (coordination)

where

```
Φ = (λ.(coordinee direction)
      (if (equal? direction '\)
        (λ.(the-pair otherdir)
          (if (and (equal? otherdir '/) (equal? the-pair coordinee))
            coordinee
            #f))
        #f))
```

We have arbitrarily chosen for Φ to apply left first. We return another λ-signature rather than a standard rightward application signature because λ-signatures cannot participate in any other combinations, e.g., composition or another coordination,[11] so this excludes "Mary slept and but or wept". The semantics of any function `coord` are very interesting, but will be deferred for now. If two adjacent categories both have functions as signatures, then Sepia will try to create new categories by applying each to the other.

Matching pairs in our lexicon, e.g., parentheses and quotes, also use the λ-signature. We would generally like sentences to have one left parenthesis per right parenthesis, to nest within each other rather than crossing boundaries, to be able to transmit the semantics of the quoted thing while having a chance to apply their own semantic piece, and to be able to surround anything. One construction is as follows:

```
(20)  ")" := close-round-parenthesis : 'close-round-parenthesis
      "(" := (λ(signature direction)          : (λ.sem (λ.close sem))
               (if (equal? direction '/)
                 `(,signature / close-round-paren)
                 (ccg/fail-this-parse!)))
```

This construction leaves `sem`, the semantics of the phrase surrounded by parentheses, unchanged, but the path has been lain to modifying it.

While it is only rarely made explicit like this, the function-based formulation of signatures allows us to keep elegantly in the lexicon some solutions that would otherwise have forced us to change the grammar, or break the abstraction barrier between parser and lexicon. We encourage sophisticated readers to think of the usual slashed signatures as common special cases of this mechanism. Signatures are then merely heuristic type-check functions to speed up processing; and the fundamental constraint on syntax is that these functions can neither check nor modify the semantics that accompany them.

3.5 Semantic errors

Because our semantics are in Scheme, and may execute any Scheme program, they may also fail in various ways. A simple example might be if someone manually entering a semantics misspelled a variable name, and the interpreter signalled an "unbound variable" error. A deeper example might be if a semantics expected to combine with a certain Scheme type but found another, e.g., from the zipcode example, if a Scheme number were somehow to get the signature digits-string, then raising it to zipcode would cause an error. Finally, again from the zipcode example, a semantics may request that the current parse be abandoned.

[11] The only reason that a λ-signature cannot participate in another coordination is that the test `(equal? the-pair coordinee)` is bound to fail. Procedures equal only their memory-obect-identical selves.

In the first two cases, with inadvertent error, the parse is abandoned and a warning is issued, hopefully leading the lexicon developer towards the bug. In the last case, the `fail-this-parse!` function causes the parse to likewise be abandoned, but silently. In every case, parsing continues without the failing combination, which becomes most important when an algorithm, rather than a human, is constructing new semantics.

3.6 The chart parser

The central data structure to the tokenizer, the parser, and the final external application that uses the parses is the parse chart. A chart is effectively a three-dimensional array—the first represents the starting position of a constituent within an input string, the second represents the ending position, and in the third are stored all of the possible parses for that interval. The tokenizer builds a chart rather than a sequence so that the parser may easily consider multiple tokenizations simultaneously. The lexical recognizer recognizes each item from the tokenizer's chart, putting into each string's place the associated category. The core CKY parser fills in the possible combinations, respecting Eisner normal form [Eisner, 1996] to avoid spurious CCG ambiguity. The output is the chart itself, so the client application may continue to process the ambiguity. A simple example parse with a chart was shown in Figure 1.

3.7 Summary of Contributions

Our new contributions to the state of the art in semantic parsing with CCG include:

- an embedding of natural language semantics into the programming language Scheme
- simultaneous single-token and multi-token lexical recognition (3.3)
- lexical recognition with regular expression patterns (3.3)
- a new combinator, adjacent functional application, that distinguishes tight from loose separation of tokens in alphabets that support it. In English text, this distinguishes tokens separated by space vs. tokens those not separated, as for time notations and telephone numbers (3.4.5)
- semantics executed at each use of a combinator, each of which may fail, which allows the semantics to influence the syntactic parse by abandoning some possible parses early (3.5)
- a view of signatures as functions that moves the implementation of CCG's Φ operator from the status of a component of syntactic theory down to being just another lexical item, and allows other useful syntactic tricks (3.4.6)

These contributions are part of the larger contribution: the most expressive semantic framework available, in which any implementable theory of semantics can be readily modeled. We move to describing our implementation of learning in this framework, both for discrimination in the face of ambiguity, and for acquiring new lexical items.

4 Learning

Ambiguity is pervasive in language, so a way to learn which analyses for a phrase or sentence are more likely than which others is critical to the success of a practical system. Linear models have been shown to

be effective in natural language parsing [Clark and Curran, 2007] and offer a seamless way to integrate local features with global ones. Perceptron-based linear learning algorithms in particular are *online*—learning happens example by example, as opposed to maximizing performance over all examples simultaneously. This means that we can construct some biases or prior knowledge manually, and integrate that seamlessly with knowledge and biases that we learn in the course of training. In principle, as better learning methods emerge, a perceptron-based linear model can easily be swapped out for another learning approach.

When we speak of knowledge in Sepia, we refer to lexical items, which give possible interpretations to sentences and phrases. When we speak of biases, we refer to the weights on each feature that allow us to choose between possible interpretations. In the previous section we discussed how to make possible interpretations from lexical items and an input string. In this section we discuss how to make features from those possible interpretations, and how to learn weights for those features. We will leave to the next section the discussion of how to learn new knowledge—how to hypothesize new lexical items.

We will assume in this report the existence of training data or *supervision*—correct interpretations for some input text—whose source and exact nature we will elaborate on in research papers citing this report. In this section we will focus on the general nature of this training data, and in particular what our method uniquely allows us to do: to learn knowledge in Scheme with semantic supervision. Learning is commonly considered in the case where the supervision specifies exactly what program is to be learned for a given phrase. By contrast, here we will focus on the weaker-supervision case where supervision only specifies evidence (z_i in Figure 2) that the learned Scheme program must be consistent with.

4.1 Perceptron-Based Linear Learning

We frame parsing in a log-linear model similar to several other approaches. In particular, the algorithm we use is adapted from Zettlemoyer and Collins [2007] and is given in Figure 2. Taking input sentences x, a parser will use lexicon Λ to produce all possible parses $\text{GEN}(x,\Lambda)$. We denote a particular parse y, and can find its semantics z using the function $\text{eval}(y)$. We define a d-dimensional feature vector $\mathbf{f}(x,y) \in \mathbb{R}^d$ to represent an input x and its parse y, and a corresponding parameter vector $\mathbf{w} \in \mathbb{R}^d$. The optimal parse for a sentence x under parameters $\mathbf{w}$ and lexicon Λ is then defined as

$$(21) \quad y^*(x) = \text{argmax}_{y \in \text{GEN}(x,\Lambda)} \mathbf{w} \cdot \mathbf{f}(x,y)$$

Note that the vector dot-product in 21 implies that the value to be maximized is the *sum* of the values in the weighted feature vector $\mathbf{w} \cdot \mathbf{f}$.

As it is described above, $\mathbf{f}$ can include features that are sensitive to arbitrary substructures within the pair $<x,y>$. We will extend the model, defining a context structure c which contains additional information about the prior sentences parsed within a discourse, and correspondingly define $\mathbf{f}(c,x,y) \in \mathbb{R}^d$ to represent an input x in some context c and its parse y, which allows features sensitive not only to the particular parse, but also to correlations and coreferences with previous parses. The optimal parse is now defined as:

$$(22) \quad y^*(c,x) = \text{argmax}_{y \in \text{GEN}(x,\Lambda)} \mathbf{w} \cdot \mathbf{f}(c,x,y)$$

Assuming sufficiently local features in $\mathbf{f}$ arising from the pair $<x,y>$, we can find y^* using dynamic programming and beam search. The features from the context c are sufficiently global with respect to the particular sentence x as not to affect the applicability of dynamic programming. Training a model of this form means learning the parameters $\mathbf{w}$, and the lexicon Λ.

We have made two other minor extensions: 1. our algorithm checks for an existing correct parse which was not best, and only generates new lexical items if no correct parse was available, as a last resort; and

For T=several iterations
    On each input $x_{i \leq n}$ and its supervised meaning z_i
        Check correctness:
            Let $\hat{y} = \text{argmax}_{y \in \text{GEN}(x_i, \Lambda)} \mathbf{w} \cdot \mathbf{f}(c_i, x_i, y)$
            If $\text{eval}(\hat{y}, c_i) = z_i$, then good: next!
        Update the lexicon if necessary:
            If $\{y' \text{s.t.} y' \in GEN(x_i, \Lambda) \text{ and } \text{eval}(y', c_i) = z_i\} = \emptyset$
                Let $\lambda = \Lambda \cup \text{GENLEX}(x_i, z_i)$
                Let $y^* = \text{argmax}_{y \in \{y' \text{s.t.} y' \in GEN(x_i, \lambda) \text{ and eval}(y', c_i) = z_i\}} \mathbf{w} \cdot \mathbf{f}(c_i, x_i, y)$
                Update $\Lambda = \Lambda \cup \{$ lexical entries in y^* $\}$
            Otherwise
                Let $y^* = \text{argmax}_{y \in \{y' \text{s.t.} y' \in GEN(x_i, \Lambda) \text{ and eval}(y', c_i) = z_i\}} \mathbf{w} \cdot \mathbf{f}(c_i, x_i, y)$
        Update the Parameters:
            Let $\hat{y} = \text{argmax}_{y \in \text{GEN}(x_i; \Lambda)} \mathbf{w} \cdot \mathbf{f}(c_i, x_i, y)$
            If $\text{eval}(\hat{y}, c_i) \neq z_i$ then update $\mathbf{w} = \mathbf{w} + \mathbf{f}(c_i, x_i, y^*) - \mathbf{f}(c_i, x_i, \hat{y})$

Figure 2: Adapted algorithm from Zettlemoyer and Collins [2007]: x = input, y = parse, z = semantics, c = context, Λ = lexicon, $\mathbf{f}(\mathbf{c}, \mathbf{x}, \mathbf{y})$ = <context,input,parse> → feature vector, $\mathbf{w}$ = the weights to be trained. The GEN function generates parses (it is the parser), whereas the GENLEX function generates new lexical items that will let GEN create the target parse result z_i from the input x_i. The principal difference from Z&C is the addition of discourse context as input to the feature function $\mathbf{f}$.

2. we modified the algorithm to allow more than one best parse if several parses' scores are identical, but for simplicity this is not shown in the figure.

In making a choice among learning methods, we looked primarily for versatility in terms of features—maximizing the information we can represent—and the ease of combining existing knowledge and biases with that gathered from training data. In joint work with Alexey Radul (unpublished), we attempted to create a Bayesian generative model for CCG parsing with Scheme semantics, but invariably found that the models were too simple to represent many of the interesting phenomena (e.g., the role of context) and at the same time too complex to train efficiently. Among discriminative methods, linear methods offer the fewest restrictions on the form of the inputs, in that features can be arbitrary descriptions combining information from the input in arbitrary ways. Among linear methods, the perceptron algorithm is appealing because its online nature allows us to potentially hand-enter some information, train another part with very strong supervision, and then continue training the same model with more abundant but weaker supervision.

4.2 Local and Contextual Parse Features

Scoring in linear models depends entirely on features for a given candidate parse. The features we make available include *inside* features about the lexical items, *local contextual* features about the sentence that a particular parse appears in, and *long-distance contextual* features about parses not necessarily in the sentence that could influence the current parse.

The local features:

- the pattern recognized for each category

- the string in the input that was matched
- the signature of each category
- the semantics of each category—if the semantics is a function, then a string representation of its source.
- the combinator most directly used to create each category

The local contextual features:

- whether the category begins at the beginning of sentence
- the number of tokens subsumed by the category
- whether there are any available categories that subsume the category
- whether the category is a child of another

On the surface, it may be appealing to add features about the nearby parses under consideration, but it is neither useful nor practical. Such features would be redundant with the actual parsing happening: those influences should be captured by lexical items that combine adjacent parses into larger ones. More importantly, doing so would break the "sufficiently local" nature of the features from the parse that allow dynamic programming.

We have experimented with some global features for coreference resolution, including:

- whether the category has a compatible signature with each possible prior coreferent
- to what extent the category has indications of similarity with each possible coreferent, e.g., the same first name or gender.

Other possible global features to experiment with for coreference resolution might be between a candidate reference and a candidate antecedent:

- whether one is a substring of the other
- the length of the longest common subsequence
- the signatures

4.3 Supervision

Let us look more deeply into the eval function, which maps a parse onto a semantics. What this function has to do depends on the form of the training data. The amount of semantic detail in our parses is very high, so we can expect it to matter how well that matches the level of semantic detail in the "true" annotations.

If the task is to parse spelled-out numbers, then the training data for twenty-three might have one of several levels of semantic detail:

(23) (+ (* 2 (exp 10 1)) (* 3 (exp 10 0)))

(24) (+ 20 3)

(25) 23

The first choice includes information about our decimal number system, and is in some sense the most informative. The second still takes apart the meaning in terms of the words, and so it is more helpful than the last, which gives only the result.

Likewise if the setting is the ATIS flight reservation domain, “show me information on american airlines from fort worth texas to philadelphia”, correct semantics might perhaps be given as:

(26) (.new FlightSearch #:from 'DFW #:to 'PHL #:airline 'AA)

(27) (λ.x (and (airline x “american airlines”) (from x “fort worth”) (to x “philadelphia”)))

(28) AA1776

The first choice, 26, contains the most information about the structure of the query, having disambiguated everything. The second, 27 is the easiest to map from the sentence, but much may be hidden in the function (airline x “american airlines”): you would expect (airline x “AA”) to return true in the same set of cases. This is roughly the level of supervision in semantic role labelling tasks. In 28 we have an example flight fitting the description but many other descriptions may match this flight.[12]

In named entity understanding, the example “Georgia” in “she went to Georgia” (but not in “Miss Georgia said” or “Georgia raised taxes”) might be marked:

(29) (.lookup Country #:name “Georgia”)

(30) Geo-Political Entity; role=Location; name=Georgia

The first specifies an exact entry from a knowledge base, whereas the second describes it. This example was of course chosen to highlight the fact that “Georgia” the GPE is not unique.

Finally, in the domain of time, let's take as an example “next Wednesday” relative to Tuesday, August 12th, 2008, which we will refer to as *ref*, or the reference time:

(31) (λ.ref (day-of-week-in (successor (week-of ref)) 'Wednesday))

(32) (λ.x (and (year x 2008) (week x 4) (day-of-week x 3)))

(33) 2008-Aug-20

The first shows how to turn the reference date into the target date, the second lets us break down the target date into its most meaningful pieces for this expression, and the last specifies the value of the target date.

In all of the examples, the highest level of semantic detail is much easier for a program to generalize than the lowest, but the lowest is substantially easier for human annotators to agree on, and is thus far more readily available. However the semantics we are proposing in our lexical model much more closely approximate the highest level of semantic detail. The central question in our learning problem then is: **Can we learn to discriminate between detailed semantic options on the basis of true annotations that have very little detail?**

Expressed differently: the shallower the semantics in the training data are, the more the choice of deep semantics for each expression becomes a hidden variable to be recovered: can we recover it using machine

[12] While it is true that the result applies only in a particular context, and the same program will evaluate to a different result at a time when different flights are available, we can bypass that complication with the assumption that we have captured, in our training input, all the relevant pieces of context.

learning and only shallow data? If not, how much deep supervision is required, in proportion to shallow supervision?

This report does not attempt to answer these questions, but simply to lay out the framework in which it is possible to ask them.

4.4 Feature Selection

The perceptron algorithm, in the limit as the number of training examples increases, weights each feature according to its usefulness: features that indicate a good parse will have positive scores, and features which indicate a bad parse will have negative scores. When a feature is unhelpful in discriminating good from bad parses, its weight will tend towards zero so as not to affect the value of the sum to be maximized.

Recall from Figure 2 that the perceptron algorithm is error-driven—that it updates only when it predicts something which turns out to be inconsistent with the supervision. Unfortuately, this allows some feature weights to have low value even when the associated features are important, simply because those features may not have much competition. To address that, it is possible to specify a minimum total value that a good parse needs to have before we will no longer bother to strengthen its component feature weights. This strategy also increases robustness in the face of noise, as described by Khardon and Wachman [2007].

Those features that have small values after training may be examined and optimized away for increased test efficiency.

4.5 Summary

We have introduced minor variations on an existing perceptron-based linear learning algorithm, highlighted its strengths and shortcomings, and detailed how scores are computed for some given parse. We have outlined the role of GENLEX, the component that hypothesizes new semantics, but we have not yet explored how it might function. That is beyond the scope of this report, and a matter for future work.

5 Perspective

We have proposed a model of semantic CCG parsing using Scheme as the language of semantics. This is an enormously flexible system that can capture any semantic model proposed to-date. We have described a way to apply log-linear learning to the parses, making use of token-local, sentence-global, and context-dependent features, the result being a powerful, introspectible, and flexible online training algorithm. Together the tools published herein serve as the basis for improved information extraction that enables more powerful reasoning about the meanings of natural language texts.

References

J. Baldridge and G.J.M. Kruijff. Coupling CCG and hybrid logic dependency semantics. *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 319–326, 2002.

R. Bar-Haim, I. Dagan, B. Dolan, L. Ferro, D. Giampiccolo, B. Magnini, and I. Szpektor. The Second PASCAL Recognising Textual Entailment Challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, pages 1–9, 2006.

P. Blackburn and J. Bos. Computational semantics. *Theoria*, 18(1):27–45, 2003.

D.G. Bobrow and T. Winograd. An overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

J. Bos, S. Clark, M. Steedman, J.R. Curran, and J. Hockenmaier. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING'04)*, pages 1240–1246, 2004.

S. Clark and J.R. Curran. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics*, 33(4):493–552, 2007.

I. Dagan, O. Glickman, and B. Magnini. The PASCAL Recognising Textual Entailment Challenge. *LECTURE NOTES IN COMPUTER SCIENCE*, 3944:177, 2006.

Christine Doran and B. Srinivas. Developing a wide-coverage CCG system, 1997.

Jason Eisner. Efficient normal-form parsing for combinatory categorial grammar. In *Proceedings of 34th Annual Meeting of the Association for Computational Linguistics, Santa Cruz*, 1996.

Lisa Ferro. TERN evaluation task overview and corpus. In *Time Expression Recognition and Normalization Workshop*, 2004.

D. Giampiccolo, B. Magnini, I. Dagan, and B. Dolan. The Third PASCAL Recognizing Textual Entailment Challenge. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, pages 1–9, 2007.

Text Encoding Initiative. P5 guidelines, 2007.

Michael B. Kac, Alexis Manaster-Ramer, and William C. Rounds. Simultaneous-distributive coordination and context-freeness. *Comput. Linguist.*, 13(1-2):25–30, 1987. ISSN 0891-2017.

R. Khardon and G. Wachman. Noise Tolerant Variants of the Perceptron Algorithm. *The Journal of Machine Learning Research*, 8:227–248, 2007.

Edward D. Loper and Steven Bird. The natural language toolkit http://nltk.sourceforge.net/, 2008.

Alexis Manaster-Ramer. Subject-verb agreement in respective coordinations and context-freeness. *Comput. Linguist.*, 13(1-2):64–65, 1987. ISSN 0891-2017.

NIST. NIST 2005 automatic content extraction evaluation results. http://www.nist.gov/speech/tests/ace/2005/doc/ace05eval_official_results_20060110.html, Jan. 10th 2006.

NIST. NIST 2007 automatic content extraction evaluation official results. http://www.nist.gov/speech/tests/ace/2007/doc/ace07_eval_official_results_20070402.html, May 2007.

Stephen G. Pulman. Formal and computational semantics, a case study. In Harry Bunt Jeroen Geertzen, Elias Thijsse and Amanda Schiffrin, editors, *Proceedings of the Seventh International Workshop on Computational Semantics: IWCS-7*, 2007.

Walter J. Savitch. A formal model for context-free languages augmented with reduplication. *Comput. Linguist.*, 15(4):250–261, 1989. ISSN 0891-2017.

Dorai Sitaram. Teach yourself Scheme in fixnum days, 2004. URL `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`.

Mark Steedman. *The Syntactic Process*. MIT Press, 2000.

Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. PhD thesis, Massachusetts Institute of Technology, 1971.

Terry Winograd. What does it mean to understand language? *Cognitive Science*, 4(3):209–242, 1980.

J.M. Zelle and R.J. Mooney. Learning semantic grammars with constructive inductive logic programming. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 817822, 1993.

Luke S. Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *Joint Coference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007 Prague)*, pages 678–687, June 2007.