The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science
M.Sc. Program - Research Track

Combinatory Categorial Grammar for Hebrew

by
Erez Agami

M.Sc. dissertation, submitted in partial fulfillment of the requirements
for the M.Sc. degree, research track, School of Computer Science
The Interdisciplinary Center, Herzliya

June 28, 2020

This work was carried out under the supervision of **Prof. Reut Tsarfaty** (Bar-Ilan Univ.) and with the help of **Dr. Ilan Gronau**, faculty contact, from Efi Arazi School of Computer Science, The Interdisciplinary Center. Herzliya.)

Abstract

Semantic Parsing is a key technology which is used to map Natural Language utterances to a formal representation of their meaning. It is employed in many modern NLP applications such as Amazon's Alexa, Apple's Siri and more. Combinatorial Categorical Grammar (CCG) is a formalism used for developing semantic parsing technology. CCG treebanks and parsers were created for many languages, but were never developed for Semitic languages. The goal of this work is to create the first CCG treebank and parser for Modern Hebrew. We use the Hebrew constituent treebank [37, 19] as a starting point for our CCG treebank.

The conversion process we propose is similar to the process proposed by Hockenmaier [13] for creating the English CCGbank. There are 3 stages to our conversion. (1) Determine each constituent type; (2) Linearize the derivation order by converting the constituent tree into a binary constituent tree. (3) Convert each constituent into CCG categories based on its location on the binary tree, syntactic role annotations and part-of-speech (POS) tags.

The Hebrew treebank contains annotation for morphemes, not words. A morpheme is a unit of meaning smaller than words. In morphological rich languages such as Hebrew, a single word is composed of multiple morphemes, and each morpheme has its own CCG category. The baseline parser that we propose is similar to Hockenmaier's head-driven parser. We use supertagging-based parsing as was shown to produce state-of-the-art results in other languages.

Supertagging Hebrew text requires morphological disambiguation. We tested three methods to apply supertagging jointly with morphological disamiguation. (1) We execute RNN on words to get a word-context-vector, then each word is split into a disambigaued sequence of morphemes. The morphemes pass-through an embedding layer and are concatenated with the word-context-vector for the generation a morpheme-word-context-vector (Word-Context); (2) We generate a lattice graph per word, each path is identified by the morphemes that are part of it. The paths pass-through an embedding layer to generate lattice-vector per word (Lattice-Encoding); (3) We created a special regional encoding for each word and its morphemes (Manual-Morph-Encoding).

Our word-context model performs best, followed by the manual-morp-encoding model. There results suggests us that the learned embeddings are better in capturing context compared to our own hand-crafted and linguistically motivated regional encoding. The lattice-encoding model performs the worst due to the enormous search space. We conclude that morphological analysis prior to supertagging is beneficial, at least in the case of our relatively small Hebrew CCG dataset.

Acknowledgements

I would like to say a special thank you to my supervisor Prof. Reut Tsarfaty for her guidance and professional support during the writing of this masters thesis. Reut shared her vast knowledge in linguistics and NLP, without which this work would have not come together. Thank you to Dr. Ilan Gronau for his support and his understanding. Thank you to Prof. Julia Hockenmaier, who shared her expertise in the construction of the German and English CCG treebanks. I would also like to acknowledge Stav Klein for helping me analyze and improve the treebank and its parsers.

Contents

List of Figures

List of Tables

List of Abbreviations

CCG	Combinatorial Categorial Grammar
CG	Categorial Grammar
HMM	Hidden Markov Model
HPSG	Head-Driven Phrase-Structure Grammar
LSTM	Long Short-Term Memory
MRL	Morpholgically Rich Language
NN	Neural Networks
NLP	Natural Language Processing
RNN	Recurrent Neural Network
POS	Part-of-Speech
PCFG	Probabilistic Context-Free Grammars
SRT	Semantic Representation of Text
NP	Noun Phrase
VP	Verb Phrase
PP	Proposition Phrase

Chapter 1

Introduction

1.1 Motivation

Imagine working on a computer software that processes text coming from a written or spoken source. The program needs to understand the information being transmitted. So we need a way to represent the meaning of the text to a computer. This representation should be easy to work with while allowing you to make inferences about the content. There should be a way to translate the raw text into that meaning representation.

In order the represent the meaning of texts formally to a computer, there is a need for Semantic Representations of Text (SRTs), as natural language is informal and ambigous, which is not a good way to represent meaning for a computer. It is ambiguous on multiple levels. SRTs are formalisms which are used to represent meaning unambigously for computational devices and downstream tasks. While Syntax dictates the arrangement of words and phrases into sentences, it does not handle all phrase-level semantics composition and semantic ambiguity. Combinatory Categorial Grammar is a type of SRT, which allows one to tie syntax and semantics in a formal and elegant fashion.

Natural language processing tasks are becoming very popular nowadays. They empower virtual assistants such as Apple's Siri or Amazon's Alexa. They are part of algorithms that try to improve health care by examining medical records. Combinatory Categorial Grammar (CCG) can be used to empower and improve many of these tasks.

1.2 Challenges

In this work we aim to adopt the CCG formalism as the basis for semantic parsing, and for the first time apply it to Modern Hebrew. We aim to create a Hebrew CCG treebank, and a statistical CCG parser trained on it. There are non-trivial linguistic decisions when applying an existing formalism to a new language. For Modern Hebrew we need to decide how to handle predicates and their subjects for the same sentence and how to represent Gender and Numeric agreement between predicates and their subjects.

Modern Hebrew is a Morphologically rich language. Morphology introduces word level ambiguity and requires us to split words into morphemes. The morpheme split sentences form a directed cyclic graph (DAG) for disambiguation. Modern CCG parsers use supertaggers based on Recurrent Nerual Networks (RNNs) for both accuracy and fast parsing times. RNNs are neural networks that remember their state which allows them to make a decision based on context around each word.

We cannot apply traditional RNN on the disambiguation of a Directed Acyclic Graph (DAG). So we need to find ways to encode the lattice and feed it into an RNN asupertagging architecture.

Finally, high word-level ambiguity will increase the interpretation search space. Moreover, the Hebrew treebank only contain 5200 sentences, a small fraction of the English and German treebanks, so training an accurate statistical model for choosing the correct interpretation presents a profound challenge, which we aim to address in this thesis.

1.3 Overview

In this work we will present the first Combinatory Categorial Grammar treebank for Modern Hebrew. The work will introduce the treebank construction process and review the different constructs of the Hebrew language. We will review several parsers (head-driven, A* bestmd, A* word-context, A* manual-morpheme-encoding) and report their results on our treebank. This thesis' manuscript is organized as follows: Chapter 2 reviews related work on CCG treebanks and parsers in other languages. Chapter 3 will focus on the Hebrew language, it will also summarizes the key features of CCG and the Hebrew language. Chapter 4 explains our conversion process and how we handle different language constructs. Chapter

5 will review our proposed tagging and parsing models, and their empirical results on our treebank. Finally, in Chapter 6 we conclude with suggestions for further research.

1.4 Contributions

This work's main contribution is the CCG treebank for the Hebrew language. Our second contribution is our super-tagging architecture suitable for morphologically rich languages. Finally, our last contribution is a baseline result for CCG Parsing over the Hebrew CCG-Bank we deliver. In future work, we suggest to increase the linguistic coverage of our treebank and examine how such linguistic enhancements improve parsing accuracy. In addition, we propose to examine the parser in the context of downstream semantic tasks..

Chapter 2

Background

In this work we address the challenge of devising the first semantic parsing benchmark and algorithm for Modern Hebrew texts. In this section we elaborate what is semantic parsing (2.1), and what are common semantic representations in the community (2.2). Then we introduce our selected formalism, Combinatory Categorial Grammar (2.3), which will be used through the thesis, and describe it in detail. We then proceed to discussing parsing algorithms (2.4) and the survey different ways we can statistically induce a parsing model using machine learning and deep learning techniques (2.5), any of which we experiment with and build upon in our empirical investigation. Finally, we discuss the specifics of CCG parsing and supertagging (2.6 and 2.7 respectively) as they will guide our designs choices in constructing the Hebrew CCG parsing alternatives.

2.1 Semantic Parsing

The goal of Semantic Parsing is to map an unstructured utterance in a language into a formal Semantic Representation of Text (SRT). Semantic Parsing is not a solved task; the state-of-the-art performance differs between languages. Languages with more speakers or with high commercial potential are better supported, while other languages might have no solution at all. A semantic parser needs to map an utterance into its SRTs target structure. Language utterances can often admit structures because of interpretation ambiguity. It is the parser's goal to select the most likely structure, that reflects that interpretation a human would assign.

Ambiguity is one of the hardest problems in parsing. Words and sentences can carry many meanings. This is unnoticed by speakers and taken for granted. We can understand the correct meaning, because we interpret a sentence in a given context. It is hard to work with a representation that has ambiguity; a good formal representation is not ambiguous. A formal representation expression should have a single interpretation that does not require its context. The parsing process therefore must determine the correct interpretation, so we can translate the utterance into an unambiguous formal representation.

Solving Ambiguity is a hard task that requires knowledge outside of the utterance context. Modern parsers use machine learning (ML) to help predict the most likely interpretation. Training these algorithms requires a treebank annotated according to the SRT (corpus). These corpora are crucial to the success of an SRT. Expert annotators built most corpora, but relying on experts is expensive and limits the corpus size. Recently, SRTs started using semi-automated and active-learning methods. These methods, in combination with modern annotation tools like Doccano [25] that integrate with active learning to improve the effectiveness of annotators by choosing the most informative samples. Crowdsourcing platforms such as Mechanical Turk are used to build high quality large corpora [31, 35].

However, crowd sourcing cannot be used when an expert knowledge is required, e.g., when assigning formal logic to represent the semantic of utterances. In this work we chose to rely on an automatic conversion process that is applied to existing resources and is automatically augmented and converted accorings to linguistic cue.

2.2 Semantic Representations

CCG is not the only method that tries to capture the semantic meaning of a text. In this section, we will briefly review other representations and methods in the field. Rappoport and Abend [2] provide an overview of these different methods.

An SRT framework provides the following functions: (1) Provide a formal notation that dictates how and which information should be represented. (2) Define rules for merging various units of meaning in its notation. In the center of all SRTs, is the identification of the predicate-argument structure (who did what to whom) and the distinction between essential (complements) and non-essential (modifiers) relations.

SRTs differ in the following ways: (1) Their interaction with syntax, some are more affected by paraphrasing while other are less affected. (2) The semantic information they cover, such as coreference, semantic-roles and spatial, temporal and discourse information. (3) How well do they handle different languages? Some were designed for a single language, while others were designed to be cross-lingual.

2.2.1 Head-driven Phrase Structure Grammar

Head-driven Phrase Structure Grammar (HPSG) by [30, 29] is a grammatical formalism to represent the correspondence between syntax and semantics, and it is similar to CCG. HPSG has its roots in phrase structure and has a rich, structured lexicon that is very similar to CCG. CCG relies solely on its categories while HPSG relies on additional information encoded as feature structures in its construction. HPSG has a multiple inheritance type system for its pre-defined linguistic signs. HPSG uses attribute-value matrices (AVMs) to represent the meanings of the signs in the language..

2.2.2 Abstract Meaning Representation

Abstract Meaning Representation (AMR) [6] does not have a strong linguistic origin like CCG or HPSG, and is more oriented to computational linguistics. AMR represents information using a rooted label graph that abstracts the syntactic idiosyncrasies (figure 2.1).

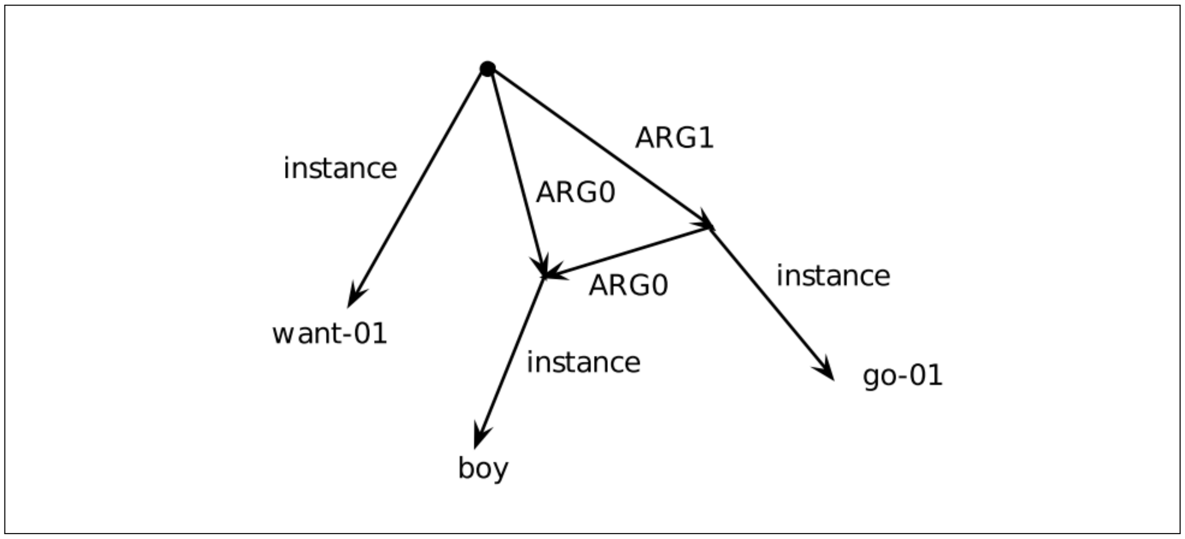


Figure 2.1: The AMR Graph for the sentence: ”The boy wants to go”. The root of the graph is the verb ”want” which receives two arguments ARG0 ”boy” and ARG1 ”go”. The matching logical form for this sentences is: $\exists w, b, g : instance(w, want_{01}) \wedge instance(g, go_{01}) \wedge instance(b, boy_{01}) \wedge arg0(w, b) \wedge arg1(w, g) \wedge arg0(g, b)$.

2.2.3 Universal Conceptual Cognitive Annotation

Universal Conceptual Cognitive Annotation (UCCA)[3, 4] is a graph-based multi-layer semantic representation. We can see an example of UCCA Sentence in figure 2.2. UCCA was designed to be cross-lingual, and it does so by abstracting away syntax more than AMR. The UCCA fundamental layer consists of 4 relation categories: Scene, Non-Scene, Inter-Scene and Other. These relations connect ”units” which can either be terminals or a collection of relations. A unit may participate in multiple relations.

The ”Scene” element is used to describe action or movement, and has, among others, 2 main types: scene with time evolution, in which the main relation is a ”Process” (P); and Scene without time evolution, in which the main relation is a ”State” (S). ”Non-Scene” is used for adjectives and other descriptive relations. ”Inter-Scene” is used for relations between scenes such as participation, elaboration and other types, marked by a relation word.

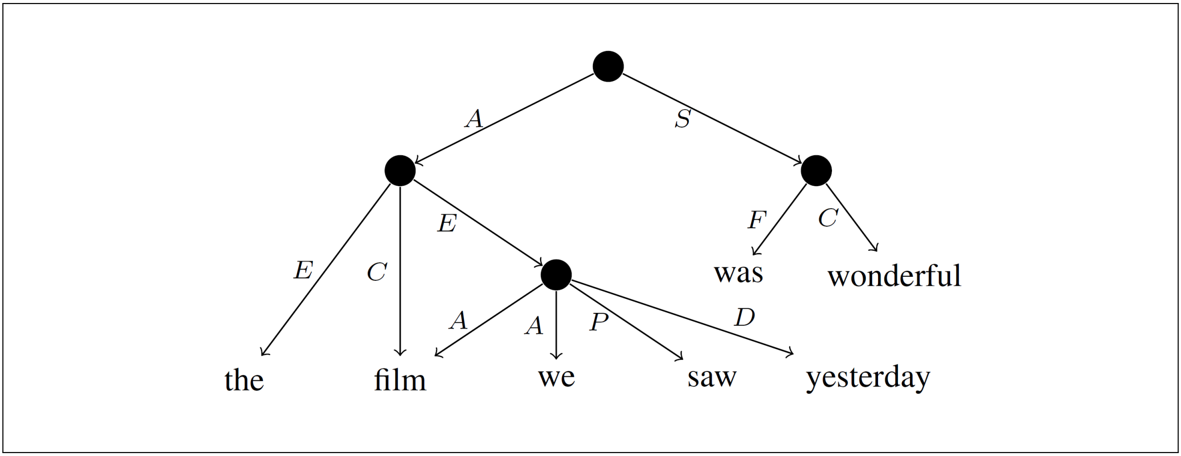


Figure 2.2: The UCCA Graph for the sentence: ”The film we saw yesterday was wonderful”. The main scene in this sentence is evoked by the word ”wonderful”. UCCA has a number of different relation types marks with the letter adjacent to the lines. For example ”P” marks process, ”S” marks state as explained before. ”A” marks participant in a scene, etc.

2.3 Combinatorial Categorical Grammar

Steedman introduced CCG [1, 32] as an extension to Categorical Grammar (CG) [9]. In Categorical Grammar we represent utterance by categories. Categories are a closed set of types which are a recursive closure of a set of atomic categories under several category construction rules. CCG is a formalism grammar that relies on combinatory logic for category construction rules. Categories are used to represent “meaning” of utterance.

2.3.1 From Utterances to Logical Forms

CCG grammar is based on combinatorial logic, which is equivalent in power to lambda calculus. CCG combiner logic can be translate rather easily to lambda calculus, thus we can translate our CCG parses into lambda calculus expressions. We map CCG categories one to one into a matching lambda calculus expression as we can see in figure 2.3.

$$\begin{array}{ccc}
\text{אני} & \text{אוכל} & \text{תפוח} \\
\hline
\mathit{NP} & \mathit{(S\backslash NP)/NP} & \mathit{NP} \\
:\lambda a.i(a) & :\lambda x \lambda y.eat(x,y) & :\lambda c.apple(c) \\
\hline
\mathit{catNP} & \multicolumn{2}{c}{\mathit{(S\backslash NP)} \quad >} \\
:\lambda a.i(a) & \multicolumn{2}{c}{:\lambda x.eat(x,y) \wedge apple(y)} \\
\hline
\multicolumn{3}{c}{S \quad <} \\
\multicolumn{3}{c}{: eat(x,y) \wedge apple(y) \wedge i(x)}
\end{array}$$

Figure 2.3: An example of CCG derviation for the sentence " אני אוכל תפוח " . In this example we include lambda calculus matching the CCG derivation. CCG assigns a category for each word. e.g: *NP* for אני and *(S\NP)/NP* for אוכל. The categories are fused resulting in a combined category according to CCG rule that will be reviewed later.

In order for the logical form to be a useful, logical expression matching to each word should be mapped to a function or an object in the relevant context. Example use cases are querying dataset or executing command (chat bots) in natural language. We do not translate our CCG parse into logical form in this work.

2.3.2 The Formalism

categories A primitive category is just a symbol, while a functional category is a function of the predicate-argument structure (which arguments it takes) matching the sign. A CCG category also includes syntactic information such as the direction of the argument. In this way, CCG links both syntax and semantics in an almost transparent way. In fact, most of the CCG complexity is in its lexicon, while the grammar itself has very few rules (figure 2.4).

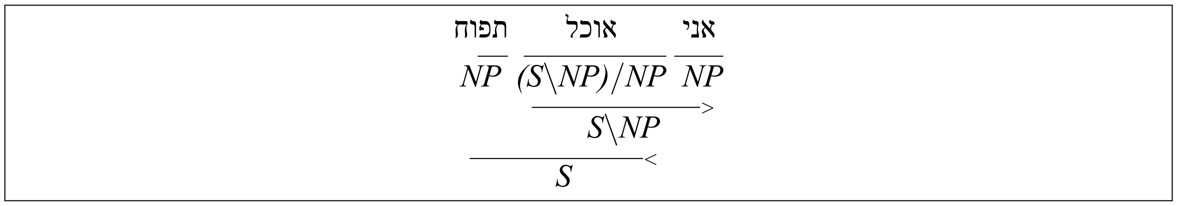


Figure 2.4: CCG derivation for the sentence: “ אני אוכל תפוח ” . The word “אני” which is a pronoun meaning ”I”, will be assigned the category NP. The word ”אוכל” (”eat”) which is a verb, will be assigned either $S\backslash NP$ (intransitive) or $S\backslash NP/NP$ (transitive). NP is the category for both “אני” and “תפוח”. NP is an example of a primitive category for noun phrases. $(S\backslash NP)/NP$ is an example of a functional category, in this case the category expects an NP object from the left and from the right.

- NP - is a primitive category used for words like: I, house, cat.
- $S\backslash NP$ - is a functional category used for intransitive verbs like: sit, eat.
- PP/NP - is a functional category used for prepositions like: in, on, at.

Figure 2.5: Example of Words with Matching CCG Categories

Application The Application rule is the most natural way to combine categories. In the application rule, a word/morpheme is combined with a word/morpheme to its left or right. We use the slash notation to show the direction of the rule.

I eat $[S/NP]$ an apple $[NP] \rightarrow$ I eat an apple$[S] >$

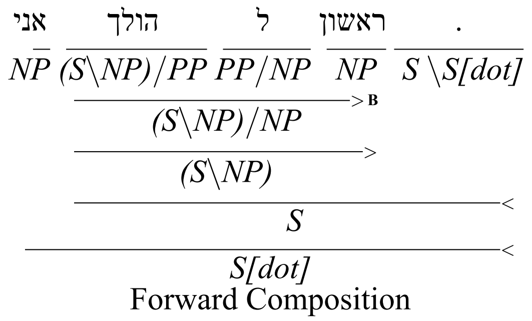
I $[NP]$ walked $[S\backslash NP] \rightarrow$ I walked $[S] <$

In the sentence ”I walked”: The category for ”I” is NP and the category for ”walked”is

$S\backslash NP$. The ”walked” will accept ”I” as an argument (backward application) resulting in the category S .

Composition Composition (figure 2.6) allows support for coordination and enables the combination of arguments in different orders. It linearizes the CCG parsing process by allowing incomplete constituents to combine, even when all their dependencies do not meet. Crossing compositions allow words to switch places. E.g. in the sentence “I walked home”. “I” category is NP and the “walked” category is $(S\backslash NP)/NP$ and the home category is NP. When we use the application rule, “walked” must first combine with “home” resulting in the category $S\backslash NP$. But composition allows “walked” and “I” to combine before resulting in the category S/NP .

- $S/PP\ PP/NP \rightarrow S/NP$ - forward harmonic $>\mathbf{B}$
- $NP\backslash PP\ S\backslash PP \rightarrow S\backslash PP$ - backward harmonic $<\mathbf{B}$
- $S/NP\ NP\backslash PP \rightarrow S\backslash PP$ - forward crossing $>\mathbf{B}_{\times}$
- $PP/NP\ S\backslash PP \rightarrow S/NP$ - backward crossing $<\mathbf{B}_{\times}$



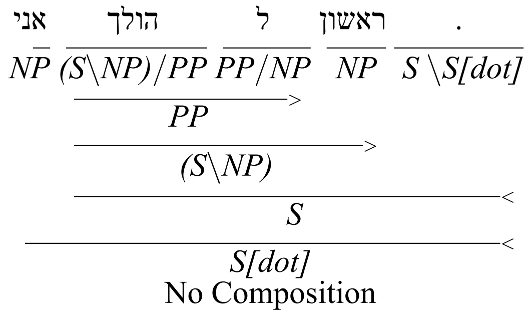


Figure 2.6: The 4 different types of composition rules. In our Hebrew treebank we only use forward and backward, as crossing composition are not needed in the Hebrew language.

Type-Raising Type-Raising turns argument categories into functional categories. It is used to maintain relation between categories in cases such as coordination (figure 2.7).

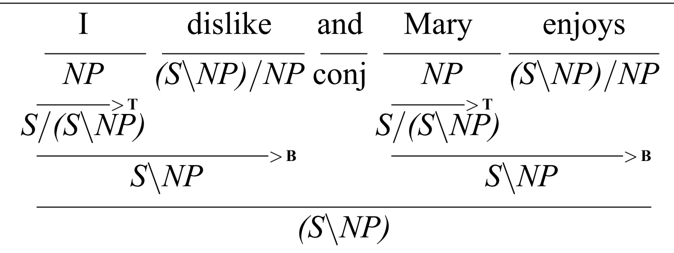


Figure 2.7: The type raising here allows ”I” to merge with ”dislike”; and ”Mary” to merge with ”enjoys” before they receive their object argument which is omitted from the example.

Hockenmaier and Steedman [13] created the CCG treebank in English. The treebank was created by converting the existing Wall Street Journal constituent treebank to CCG. They based the conversion on finding the semantic head of each constituent. There have been works on other languages including German and Arabic, which is also a semitic language.

Hockenmaier [18] worked on the German Language treebank based on the Tiger Corpus. Hebrew and German are similar in allowing more flexible word ordering compared to English. Cakici [12] converted the Turkish dependency treebank to CCG. Turkish and Hebrew have a more agglutinating structure compare to English. Boxwell et al. and Eltaher et al. [11, 34] worked on the Arabic treebank.

Arabic and Hebrew are sister languages in the semitic family, sharing many features. Both have missing vowels from their writing system, which can cause word-level ambiguity. Hebrew and Arabic have a similar root system that consists of 3 (usually) skeletal consonants. The skeleton may gain affixes: prefixes, suffixes, infixes. These affixes change the meaning of the skeleton. To our knowledge, there has been no previous work on applying CCG on the Hebrew language.

In morphologically rich languages e.g. Hebrew, Arabic, Turkish, etc.., a morphological process creates words by combining multiple morphemes with different syntax and semantic roles. We also model the morphological process in CCG as well by breaking words and assigning categories to morphemes.

2.4 Parsing

A key component for practical SRT is the parser. The parser's job is to translate a raw sequence of words into the SRT notation. The parser is not part of the SRT, and there may be multiple parsing strategies for the same SRT. A parser works incrementally, it starts by building candidates parses for the smallest unit and gradually merges them together untill it covers the complete sentence or phrase. In this process, a parser may evaluate thousands, or even millions of possible parses. Due to language ambiguity and due to some SRT attributes, a single sentence or phrase may have more than 1 valid parse.

In order to select the best-suited parse, we turn to statistics. A statistical parser selects the best candidate parse by assigning a probability to each parse and taking the most probable. A parser might evaluate all possible parses for a given sentence or phrase in any order, and select the best parse candidate out of all possible parses like the vanilla Cocke–Kasami-Younger (CKY) algorithm is doing. Another option is to try to evaluate more likely parses first, then stopping early without exhausting all possible parses. It is the strategy that is taken by A* parser. Different parsers use different heuristics on which parses they should evaluate first.

2.4.1 Cocke–Kasami-Younger

Cocke-Kasami–Younger is a dynamic programming parsing algorithm for context-free languages. Vanilla CKY calculates the parse for every possible subsequence of given phrase or sentence. It starts from subsequence at length 1 untill it gets to the length of the input. The algorithm evaluates every possible partition for subsequences that is bigger than 1, where it tries to merge two subsequences at the partition point.

2.4.2 A* parser

A* parser by Steedman and Lewis [22], works by keeping an agenda of candidate constituent categories to explore. The agenda is sorted by estimating the probability of the entire sentence/phrase parse containing the category. The probability of the parse is estimated by using the category intrinsic probability with the extrinsic probability. The intrinsic probability is estimated using the sum of the selected categories' probabilities. The extrinsic probability is calculated by summing the best possible candidate for each constituent. The parser picks the top item on the agenda to calculate possible candidates which are added to the agenda. A* parser is significantly more efficient than other parsers (CKY for e.g), so for this reason we chose to implement A* parser for CCG parsing of the Hebrew treebank.

2.5 Modeling

The objective function (equation 2.1) of the parser is to find the most suitable formal representation that matches its input. The selected formalism (in our case, CCG) determines the search space Y with is CCG case is all possible y's for a given sentence x. A probabilistic Model determines the function $P(y|x)$.

$$f(x) = argmax_{y \in GEN(x)} P(y|x) \tag{2.1}$$

2.5.1 Head Driven

Head-driven generative models grow the parsing tree from the roots to its leaves using 3 equations. This model was used by [13] for building a CCG parser in English. For this reason we implemented this model as a baseline CCG parser.

The model objective function (equation 2.5) tries to find the best CCG tree out of all possible CCG derivations for a given sentence. The Model includes 3 probabilities: it uses one to generate the head from the parent; Another to generate the other child (in binary cases); And the third to generate the word from the category. The head conditional probability models the growth of a head child from the left or right direction of its parent.

$$P_{head}(Head|Parent, exp) = \frac{Count(Parent, Head, Dir)}{Count(Parent)} \tag{2.2}$$

The non-head conditional probability models the growth of the child node to the other side of the head node, in case of binary growth.

$$P_{nonhead}(nonHead|Parent, exp, Head) = \frac{Count(Parent, Head, Dir, NonHead)}{Count(Parent, Head, Dir)} \tag{2.3}$$

The lexical conditional probability models the growth of a word from the parent category.

$$P_{lexical}(word|Category) = \frac{Count(Word, Category)}{Count(Category)} \tag{2.4}$$

Our complete probability model tries to maximize the probability of the entire derivation tree.

$$y = argmax_{y \in Gen(x)} \prod_{t} P_{head}(t) P_{nonhead}(t) \prod_{l} P_{lexical}(l) \tag{2.5}$$

2.5.2 Log-Linear

The log-linear model can estimate the probability from our feature table. Log-Linear model has a high flexibility that allows usage of many features. We base the model on the following estimation function, where each feature has it matching b_i coefficient. $y = b_1 e^(b_2 x_2 + b_3 x_3 + b_3 x_3 ..)\epsilon$
We get the log linear model by applying log function on the estimation function

$$\ln(y) = \ln(b_1) + b_2 x_2 + b_3 x_3 + b_3 x_3 ... + \ln(\epsilon) \tag{2.6}$$

We note that the coefficient $b_i = \frac{d \log(E(y|X))}{dx_i}, i = 2...n$ in Log-linear models can be trained by various methods, most commonly by maximum likelihood estimation over the log likelihood function
The Possible features can be the number of times something occurred, in the previous words, the last 1-3 letters in each word (capture present progressive, possessive) in English, and more.

2.5.3 Perceptron

Perceptron is a linear classifier with a single layer. The Perceptron can be train against different aim functions. Let $\Phi(x, y)$ be a method to extract vector of features from a pair of a sentence and a possible parse (a feature could be the number of time a rule was used etc).
A common objective function is:

$$y = argmax_{y \in GEN(x)} \Phi(x, y) \cdot \alpha \tag{2.7}$$

The parameters of the function are learned using iterative algorithm: (1) The feature value result is calculated on the input features. (2) The results are subtracted from the expected. (3) The weights are updated $\alpha = \alpha + \Phi(x_i, y_i) - \Phi(x_i, z_i)$ where z_i is an incorrect parse for which the perceptron return the highest likelihood.

2.5.4 RNN - LSTM - GRU

Neural Networks are a set of machine learning algorithms that are inspired by biological neurons. The neurons are connected with one another, and some can save their state. Each neuron multiplies its inputs by its learned weights. Then a non-linear function (sigmoid, tahn, Relu...) is applied.

Recurrent Neural Network (RNN) is a subset of Neural Networks where the neurons also use their previous states in calculation. RNN show impressive performance in some NLP tasks. Human languages are sequential in order and interpreting a word depends on words previously observed. In a simple RNN each neuron receives its previous output. Vanilla RNN suffer from vanishing gradient which limit their ability to remember long range dependencies and rarely perform well in practice. Instead, most RNN are built of more complex models with higher units called cells which contain several simple neurons. For these reasons, we implemented 3 RNN parsers, which we will review in chapter 4.

Long Short Term Memory

Long Short Term Memory (LSTM) is an RNN which are is specifically designed to avoid the long-term dependency problem. LSTM cell comprises 3 gates: forget gate (equations

2.8), input gate (equations 2.9-2.11) and output gate (equations 2.12,2.13). These 3 gates control what information to forget, keep and output accordingly. Each gate uses sigmoid as non-linear function.

The forget gate controls how much information to keep from the previous state.

$$f_t = \alpha(W_f[h_{t-1}, x_t] + b_f) \tag{2.8}$$

The input gate controls how much information to add to the cell state.

$$i_t = \alpha(W_i[ht - 1, x_t] + b_i) \tag{2.9}$$

$$\tilde{C_t} = \text{tanh}(W_c[ht - 1, x_t] + b_C) \tag{2.10}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t} \tag{2.11}$$

The output gate controls the current output of the LSTM cell.

$$o_t = \alpha(W_o[ht - 1, x_t] + b_o) \tag{2.12}$$

$$h_t = o_t * tanh(C_t) \tag{2.13}$$

Gated Recurrent Units

Gated Recurrent Units (GRU) is an RNN with 2 gates: reset gate and update gate. The reset gate (equations 2.14-2.16) controls how much previous information to forget.

$$z = \alpha(W_z x_t + U_z h_{t-1} + b_z) \tag{2.14}$$

$$\tilde{h}_t = tanh(W_h x_t + r * U_h ht - 1 + b_z) \tag{2.15}$$

$$h_t = z * h_{t-1} + (1 - z) * \tilde{h}_t \tag{2.16}$$

The update gate (equation 2.17) decides what information to add and what to forget.

$$r = \alpha(W_r x_t + U_r h_{t-1} + b_r) \tag{2.17}$$

GRUs are more computationally efficient than LSTM and tend to converge faster. For this reason, our model parsing models use GRU instead of LSTM.

Embedding and Input Encoding

Embedding is a dimensionality reduction technique that maps discrete variable to a vector of continues numbers. In NLP, common methods for generating word embedding utilise negative sampling with either a neural network or skip-gram [23]. We prefer word embedding over discrete identifier for words representation because of their meaningful representation, which is better suited for neural network nature. A navie way to represent categorical features is by generating a binary feature for each possible value (One Hot encoding). We prefer embedding over One Hot encoding since encoding one column per word will cause large and sparse vectors, which will require lots of parameters. Another drawback of One-Hot Encoding is their uniform nature, which does not give similar representation to words with similar meaning.

Embedding can be trained on unlabeled data and easily shared between different NLP tasks, which is a common practice in NLP. There are several open source frameworks to train word embeddings such as Glove [28] and Fasttext [10]. In our RNN experiment we train our own embedding together with our models rather than using pre-trained embedding. We found that pre-trained embedding resulted in performance loss for our models. We believe that it is because of our treebank morphological disambiguation. Since Pre-train embedding are available for words not morphemes, there seems that they do not offer any advantages in our case. We believe that training embedding separately from the model make sense when you have external dataset which cannot be used to train the entire model , due to lack of labels for example.

2.6 CCG Parsing and Probability Modeling

In her work on the English language, Hockenmaier [13] used a CKY head base statistical parser to parse CCG. The CKY algorithm builds the parse from the bottom up, each time merging two constituent symbols. They build the statistical model on top of the CKY chart by including the conditional probability of the two symbols given their parent symbol. A cell in the CKY chart may contain many symbols that can make actual parsing intractable. It uses a beam to select only the top N symbol from each cell at the expense of missing rare derivations.

Clark and Curran et al. [14] introduced a more sophisticated C&C parser. The C&C parser uses a log-linear model, also known as Conditional Random Field (CRF). CRF can easily include more features than just the parent and children symbols used previously. The C&C parser is more accurate and its parsing time performance is just as quick; However, it requires a time consuming training. C&C include a supertagger [20, 7] to increase the efficiency of the parser and CRF. The supertagger predicts a list of categories for each word. The list should be significantly smaller than all possible categories but big enough to include the correct category. Steedman and Lewis [22] took it a step further by using only the supertagger for statistical modeling. Their simplified parser is quicker and more accurate. Clark et al. [39] shows how using RNN supertagger results in better accuracy.

2.7 Supertagging

We use supertagging in CCG to provide lexical probabilities to words/morphemes. The supertagger provides benefits to the parser. First, it increases the parsing efficiency by dropping lexical entry (categories) with a low probability; second, it simplifies the parser's statistical model. The CCG category contains information about how, and with what other category in can merge. Thus, supertagging in CCG can solve a big part of the parsing problems.

A* parser for CCG was first used by Lewis and Steedman [22], their parser relies only on the supertagger's probability in order to predict the most likely parse. A* parser is very efficient as it does not have to maintain a complex ”book-keeping” of probabilities that is usually required for parses with an internal statistical model. It is our goal to devise both a supertagger and a parser for the Hebrew language based on the CCG formalism, working at the level of morphemes.

2.8 Conclusions

We saw here that semantic parsing is a critical task in NLP and that in order to achieve this we are required to choose a representation, construct a treebank, design a parser, and train a parsing model. We surveyed different representation options and have chosen to adopt CCG. We further survey different parsing and machine learning methods previously used for semantic parsing, and which we are going to assume as building blocks when devising our own Hebrew semantic parser. Next, in chapter 3, we discuss our data, coming from Hebrew, a morphologically rich languages, and the particular challenges it poses to CCG parsing.

Chapter 3

The Data

In this work we address semantic parsing of Hebrew, a Semitic language. Semitic languages have distinctive properties that make their processing harder than English and similar languages, in particular due to their rich and ambiguous morphology. In (3.1) we explain what is morphology and illustrate what are morphologically rich languages. In (3.2) we survey specific morphological and morphosyntactic phenomena in the Hebrew language, and in (3.3) we illustrate how these morphological-syntactic phenomena is captured in the Modern Hebrew treebank – which is currently the only existing benchmark for Modern Hebrew parsing.

3.1 Morphologically-Rich Languages (MRLs)

Morphology is the process in that governs the creation of words. Some languages have very limited morphological processes if at all, while others have very complex ones. We divide morphology into a derivation process and an inflectional process. A derivation creates new entries in the lexicon. An inflectional process encodes additional information which is required by the grammar and depends on the context of the word. In English, adding ”ing” at the end of a word is an example of an inflectional process.

Hebrew is a morphology rich language, which increase both modeling and parsing challenges, which compels us revised the English base CCG model and parser.

Hebrew has a number of inflectional processes. We inflect words according to gender, tense, number and person. Words in the Hebrew language can also contain a functional part that is not related to the words themselves. We know these as clitics. We can see an example of morpheme with a syntactic scope larger than the word itself, in the word “בבית” (in the house). It contains 2 morphemes (a preposition and a noun). The first letter “ב” (in) denote the preposition. (house) “בית” is the noun part. In this example, the morpheme carries information that is a part of the grammar and helps to assign a structure to the sentence.

We can see how morphology affect ambiguity in the word “Btselem”:

“Btselem”: noun .בצלם

“in Tselem”: preposition ב (in), noun צלם (Tselem)

“in their shadow”: preposition ב (in), noun צל (shadow), possive ם (their).

Because we attach the information to the host word, words in a phrase or in a sentence have more than one role in the semantic representation and sometimes it is difficult to draw the line between the roles of such a word.

Each word is composed of morphemes and each morpheme has a different role. It is not effective or possible to parse at the word level. Instead, the words are split into their morphemes. There may be more than one possible segmentation to the same word. Therefore , the parser has to handle word-level ambiguity. To address this, the word segmentation can be done as a separate phase of the parsing process or as an integrated phase of it (joint parsing).

Therefore; Morphology rich languages present a challenge to both SRT and parser. We use Lattices (figure 3.2) to represent the different decomposition of words into morphemes. A morphological analyzer maps the words into a morphemes graph. The morphological analyzer does not take into account syntactic or semantic rules. A path in the morpheme graph represents a possible decomposition for the words. The composition might be grammatically incorrect.

3.2 Modern Hebrew

Modern Hebrew is a semitic language spoken mainly in Israel. Hebrew, as other members of the semitic family, has a rich morphology and syntax, with fusional properties. Words in the Hebrew language are composed of smaller units of meaning called morphemes. We divide morphemes into two groups, derivational and inflection. Inflection maintains the core word's meaning while adding information such as tense, gender, and number. Derivation changes the core meaning of the word, e.g., it can turn a noun to a verb.

Hebrew has inflectional morphology governed by the grammar in a process called agreement. For example, the subject and a verb must agree on the same number and gender. A noun and its adjective must agree on the determiner marker. The inflectional morphology is not just syntactic "sugar", but plays an important role in predicate-argument structure. The inflectional morphology helps the speakers understand the relation between words (e.g. subject or object of the verb). We need this process since Hebrew has a flexible word order. SVO (subject-verb- object) is the most common word order for the Hebrew language.

Modern Hebrew proposition and case markers are inflectional and they are attached to the beginning of each word.. The Word "בבית" which is composed of the morphemes "ב" (in), "ה" (the) and "בית" (house), would translate to "in the house".In this case, the morpheme "ה" (the) does not manifest in any letter in the word-final form.

Hebrew has 7 formative letters called "משה וכלב." These always attach to the left of their host word. Their scope is a word, a clause, or a phrase. We use them in the clause level as coordinators, subordinators, relativizers. In the phrase level, we use them as prepositions and modifiers. In the word-level, we use them for determiner. We inflect nouns with gender and number. We inflect most pronouns for number and person, but some pronouns are neutral and do not include a person. We inflect verbs for gender, temporal, numeral and person.

Modern Hebrew has a multi-layered word-formation, the derivational part is based on a 3 letter root which is plugged into templates to form words. The inflectional morphology of the Hebrew language includes gender, temporal, person and numeral attributes. These attributes need to agree between words in a construction.

One of the challenges of parsing texts in Modern Hebrew is its high word-level ambiguity. This ambiguity is the result of Hebrew's rich morphology in addition to the Hebrew writing system. the modern Hebrew writing system includes diacritics (figure 3.1)

In the word "אִמָּא" the first letter אִ is pronounced "Ee".
While in the word "אַבָּא" the first letter אַ is pronounced as "Ahh"
In most texts, both words will appears without dictate as אמא and .אבא

Figure 3.1: Example of Diacritics in Modern Hebrew

marks that dictate the sound (Vowel) of written text. However, Hebrew speakers rarely use diacritics in everyday writing of the language. Speakers of the language can conclude the right word from the context.

Current works on the Hebrew language first break words into their morphological component (see figure 3.2) and only then apply a parser. Just like in the sentence level, there can be more than one way to break a word into morphemes, as the same word-final form might be created from different roots with different inflectional morphology. Tsarfaty et al. [17, 38] found that joint parsing results in better performance.

Most supertagger implementations will only work on a linear sequence. In joint parsing, where parsing and disambiguation are done in a single step, the input is a lattice graph. Linear supertagger can only work on each path in the graph separately, this is exponentially increasing and unpractical. The category of each morpheme depends on other morphemes in the graph. These dependencies require us to treat every path in the lattice graph differently. Since the number of paths increases exponentially with the number of words, the problem quickly becomes intractable. Bar-Haim et al. [8] createed a POS tagger by merging the sequence of morphemes in the word level into a superstate. They encountered similar problem [33] when trying to convert speech into text. They used a combination of RNN and HMM to provide a better model than just HMM. The HMM model selects the top paths, which are rescored by running the RNN Model.

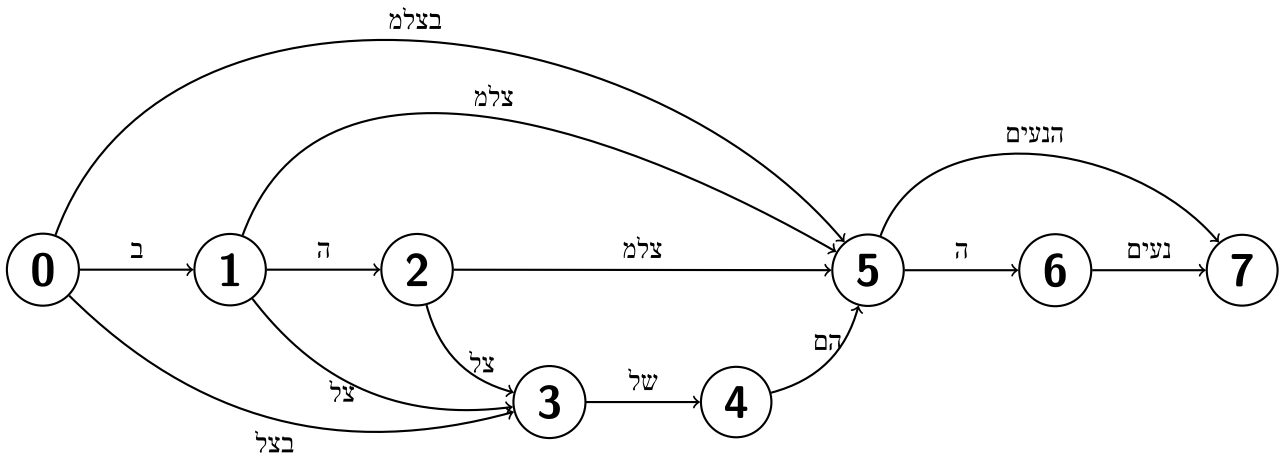


Figure 3.2: The lattice for the sentences: ” בצלם הנעים ” .
The word “בצלם” (states 0-5) and the word “הנעים” (states 5-7) are composed of multiple morphemes. We can see that there are 12 ways to decompose the two-words sentence into morpheme. An average word in Hebrew is composed of between 3 to 4 morphemes, so the number of possible paths will increase by at least 3^n where n is the number of words. This large search space requires us to find as efficient way to process the lattices.

3.3 The Modern Hebrew Treebank

Building a treebank is a large scale, time and resource consuming task. It is a common practice to convert an existing treebank from one formalism into another, thus saving a lot of effort. Hebrew has an existing constituent treebank, which we will convert into CCG formalism.

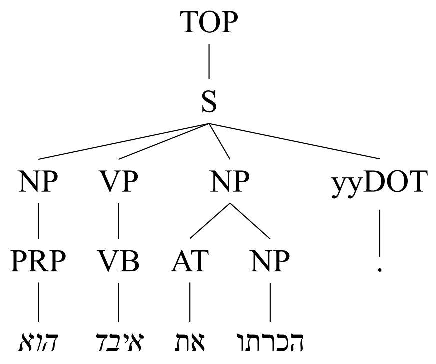


Figure 3.3: Example of Sentence from the Hebrew Treebank.

Sima'an [19] built the treebank for Modern Hebrew CFG in 2001. The treebank contained 500 hand annotated sentences taken from the Haaretz newspaper. It was later extended to 6,200 sentences by using automated, parsing with manual work. The original treebank annotation were constituent tree which included morphological feature such as gender, number, person and tense. We use a derived version of the treebank from Tsarfaty [37] which includes syntactic roles. In Figure 3.3 we can see a simplified tree from the Hebrew treebank. The lowest nodes are morphemes text. the "TOP" symbol marks the highest node in all. We omit additional information such as syntactic role and morphological features from the figure. The word ”הכרתו“ has 3 parts: "ה" (def marker) - "כרת" (consciousness) - "ו" (his) . The treebank has 3 segments: train (5,241 sentences), dev (483 sentences) and test (492 sentences). It treebank annotations include POS, syntactic role, and inflectional morphology, as noted above. The treebank uses a transliteration scheme, in which Hebrew letters are replaced by English letters.

The morphological analyzer is the first phase in analyzing a text in the Hebrew language. Adler et al. [5] present a morphological analyzer based on BGU Lexicon. We are using the analyzer and the disambiguator, Amir and Tsarfaty [24]. Bar-Haim [8] presented a POS tagger for the Hebrew language. Tagging Hebrew text is more complex because POS tags are applied on morphology and not words. They solved the problem by combining all the POS tag of a word's morphemes into a single symbol. Tsarfaty [37] presented a constituent parser based on relationship realization. Goldberg [16] worked on a dependency version of the treebank and presented a dependency parser. More and Tsarfaty[24] presented a dependency transition parser.

Chapter 4

Building A Modern Hebrew CCG Treebank

The first contribution of this thesis is a Hebrew CCG treebank, where sentences are annotated with CCG derivations. The treebank has two purposes, as train data for training the parsing algorithms, and as a benchmark for evaluating the performance of the different variants. This section presents the algorithmic conversion we devised (4.1) its linguistic coverage (4.2) and basic statistical analysis of the resulting treebank (4.3).

4.1 The Treebank Conversion

We base the tree conversion process (Fig. 4.1) on Hockenmaier [13]. It consists of 3 primary stages: (1) case marking (2) tree binarization (linearization) (3) categories construction.

As pre-phase to the constituent identification, we apply some transformation to the CFG trees (figure 4.1a-b). The attribute role 'information' helps us determine the different function each constituent node plays. Our first transformation removes unary nodes and changes constituents with special characters like dashes and parentheses. When we unify unary nodes, we take attribute role from the parent nodes, while the morphological attributes are taken from the child node. The transformation shortens the tree paths while keeping all the original information.

4.1.1 Case Marking

The head constituent is identified by rules over the constituent role attributes. For each constituent we rank the child nodes base on their rule attribute, the highest node is chosen to be the head.We determine the type-of non-head constituents after the binarization process. At that stage, the tree is the same as the final CCG derivation tree. The types are an adjunct, complement, punctuation. Complement and punctuation are marked be on a curated list of roles. The remaining nodes are marked as adjunct. The constituent type is annotated on every node of the tree, which is passed to the next phase.

4.1.2 Tree Binarization

We do a tree binarization to simplify the category creation phase. The binarization is done by splitting constituent containing over 2 nodes. The binarization determines the attachment order of our CCG trees for these constituent. We use left node-raising for nodes to the left of the head constituent, and right node raising for nodes to the right of the constituent root see (figure 4.1c). The binary tree is now the same as the final CCG derivation tree, but we do not yet have the categories at each node.

4.1.3 Categories Construction

This phase generates the final CCG tree (Fig. 4.1d) by traversing the tree from top to bottom. The inner node traversal order is determined by case marking. Complement are traversed first, followed by head and finally adjunct. The reason for this order is that complement categories do not depend on their head, but the head category does depend on the complement. Adjunct depend on their head so the head category must be determined first. Complement Category is derived directly from their POS, with a exception rules to handle FRAG and other noise in the original treebank. Adjunct Category is derived from the category of the head. We do adjunct category shortening to prevent long categories due to a chain of modifiers. Adjunct Shortening uses the inner results category of its heads, instead of the complete category. e.g: an Adjunct for NP/NP will be NP/NP instead of $(NP/NP)/\ (NP/NP)$

We calculate the categories of the head nodes at the end of each sub-tree. If the parent node contains two children (head and adjunct) the head category is determined by taking the parent category as is, if (head and complement) the head category is constructed by adding the complete to the parent category.

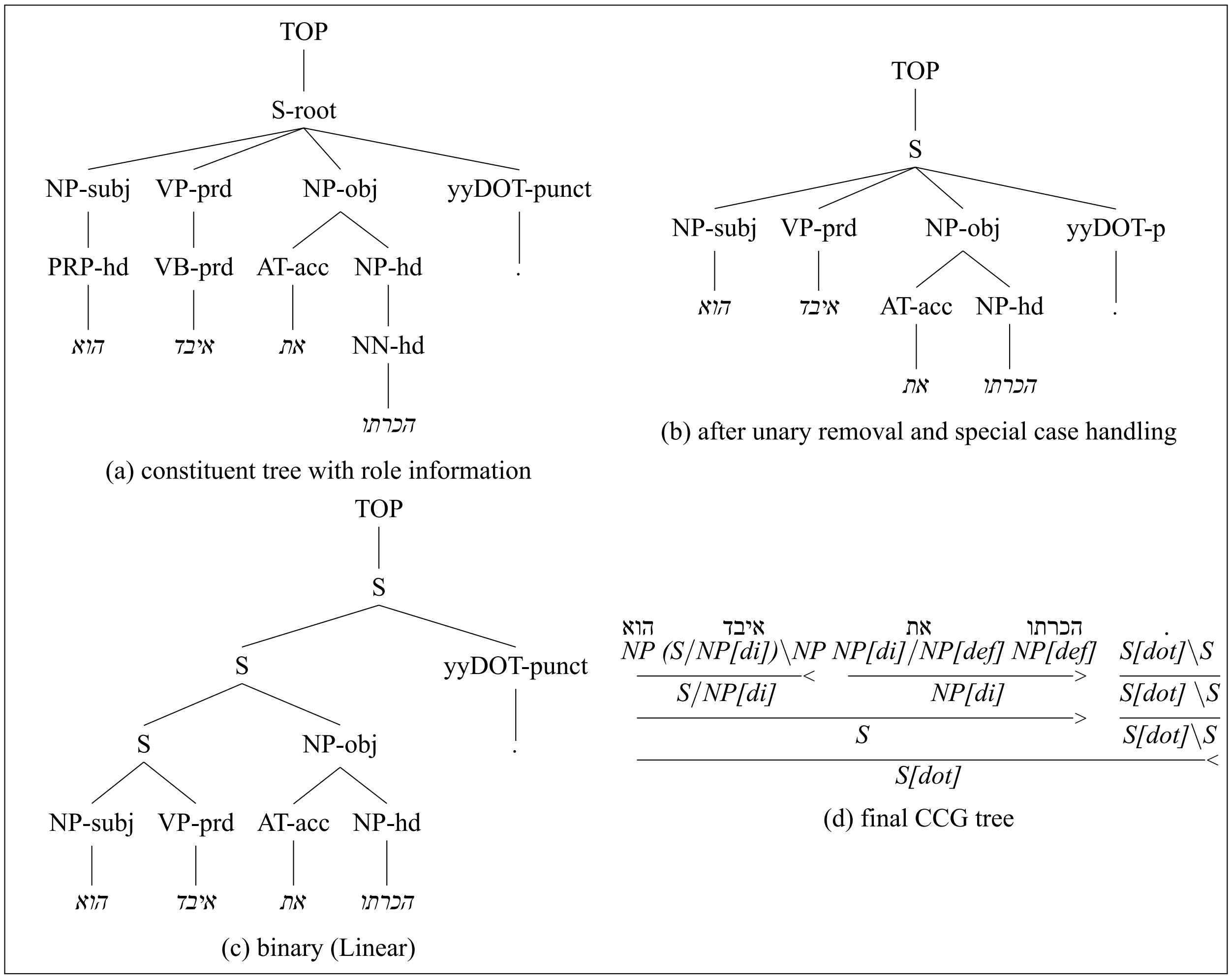


Figure 4.1: The 4 Stages conversion process: in the first stage "unary removal and prepossessing" (a-b), we can see how we remove the unary node PRP assigned to the word .הוא There are also some transformations of the tree structure for special conjunctions and constructs. The second stage "binary transformation" is shown in sub-figure c. The head of this sentence is the word איבד and the binarization process is done around it. sub-figure d shows the final CCG tree. In it we can see that the category for the word איבד receives both NP subj and NP[di] direct object as arguments. In the 4th stage, "case marking phase", we annotate each node with a type (head, complement, adjunct and punct) the head marking is done between sub-figure b-c, while the remaining types are annotated after binarization between sub-figure c-d.

4.2 Linguistic Coverage

4.2.1 Definiteness and Agreement

The morpheme “ה” express definiteness in the Hebrew language. It is similar to the word “the” in English. “ה” is always attached to the word it acts on. For example, the two morpheme word “הבית”, composed of ה and בית, translates to “in the house”. Hebrew exhibits definiteness agreement in a noun phrase. E.g. in the phrase “הבית הגדול” which translates to “in the big house”. We note that “ה” attach to both the noun “בית” (house) and the adjective “גדול”(big). It would be a malformed sentence otherwise. In our treebank, the definiteness marker ה received the category, *NP[def]/NP*.

4.2.2 Pro-drop

Pro-Drop is a case where the pronoun in the sentence is dropped. For Example in figure 4.2. The sentence *אני הלכתי לעבודה* can be written as *הלכתי לעבודה*. The pronoun *אני* which translate to “I” in English was dropped. The pro-drop affects the verb category; in our case, the verb *הלכתי* will have two categories. The category *S /NP \NP* for the regular case and *S /NP* for the pro-drop case. Pro-drop increases the lexicon size, as many verbs can appear with and without their subject. In this work, we decided not to handle pro-drop separately. This can hurt the performance of our parser for verbs that only appear in a transitive or an intransitive form. We consider introducing meta rules that will add lexicon entries for verbs with and without pro-drop.

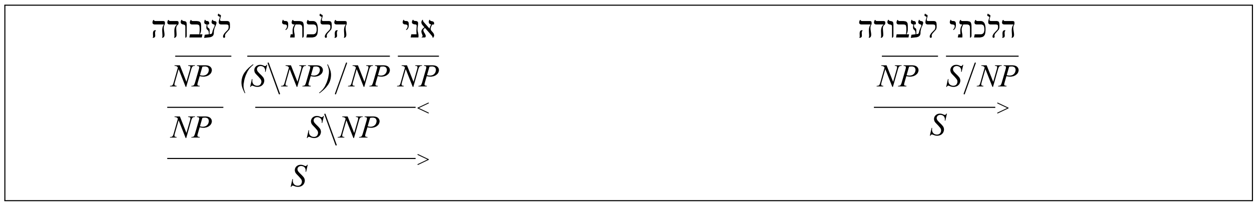


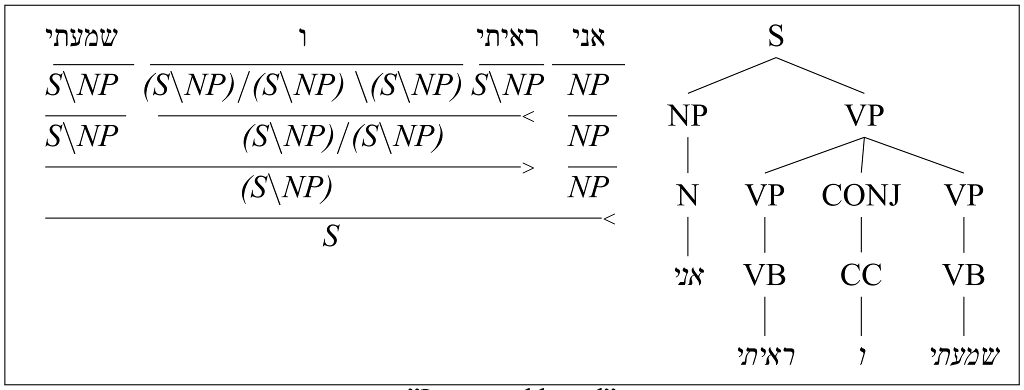
Figure 4.2: Example of Pro-drop in Modern Hebrew

4.2.3 Conjunction

We model conjunction within CCG categories. In our treebank conjunction and commas have categories just like any other morphemes. This differs from previous works which add a new rule to CCG similar to Hockenmaier [13]. This approaches allows us to treat conjunction similar to any other morphemes, it also provides lexical visibility to conjunction categories and their usage. Our approach has some drawbacks. It results in noisier categories for 2 reasons: (1) Conjunction word can combine many categories, so the conjunction word may have many categories that are semantically similar e.g.: $(NP\backslash NP)/NP$ $(S\backslash S)/S$. (2) Because we build our treebank propagating categories from lower part of the PCG tree, errors in lower categories will cause incorrect conjunction categories add more noise. In the figure 4.3 "ו" is the conjunction marker and its category is $(S\backslash NP)/(S\backslash NP)\ \backslash(S\backslash NP)$. We note that we treat the conjunction morpheme as the head of the constituent. The Morpheme is not the semantic head, but it is the syntactic head. Our decision means that we need to separate semantic and syntactic heads. Commas can sometimes accommodate the conjunction morpheme, in these cases we treat the coma as a modifier on the conjunction morpheme. In other cases, we might omit the conjunction morpheme leaving only a comma. In such cases, we treat the comma as a conjunction with morpheme. The multiple role comma plays causes its lexicon categories to be noisier than others.

4.2.4 Gender Agreement

The Hebrew language nouns are divided into feminine and masculine genders. The gender agreement in Hebrew inflects verbs and adjectives to match the subject gender. The inflection can help Hebrew speakers resolve predicate-argument structures in complex sentences. Tsarfaty [36] showed that using gender agreement can improve parsing accuracy. Adding gender subcategorization to categories will prevent merging different genders. In figure 4.4 the verb “מטרייה” (Umbrella) appears with the adjective “גדולה” (big) which is the feminine form of the adjective. We note that “מטרייה” is a feminine Noun.



”I saw and heard”
The word אני meaning ”I”
The word ראיתי meaning ”saw”
The word שמעתי meaning ”heard”
The word ו meaning ”and”

Figure 4.3: Example of Conjunction

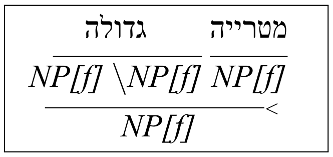


Figure 4.4: Example of Gender Agreement in Modern Hebrew

4.2.5 Flexable Word Order

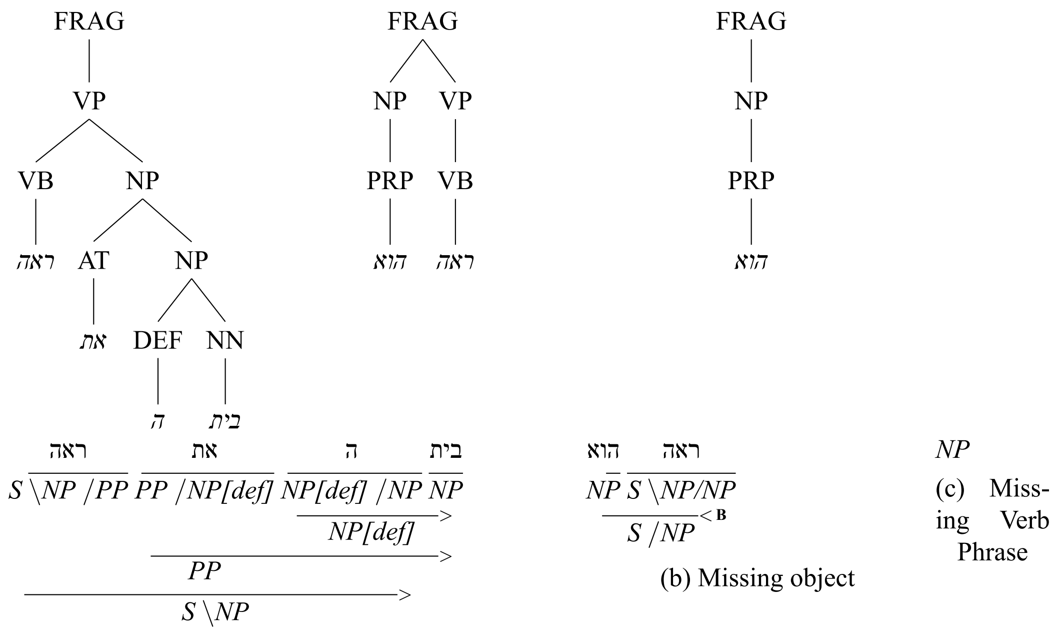
Word order in Hebrew is usually tied to focus. Hebrew’s flexible word order influence mostly verbs categories. The most common word order is SVO (subject, verb, object). A likely category for a transitive verb in that order will be *(S\NP)/NP[di]*, however, in an OSV order that verb will get the category *(S\NP)\NP[di]*. Word order changes cause verb categories to be more sparse even though they are semantically similar.

אני אוהב לטוס
לטוס , אני אוהב
אני - I אוהב - love לטוס - to fly
The first line is in SVO while, the second is in OSV and is more likely as a responses.

Figure 4.5: Example of different word order in the Hebrew language

4.2.6 Incomplete Sentences

The Hebrew treebank contains incomplete sentences. Incomplete sentences (Fig. 4.6) are missing subject (Fig 4.6a) or object (Fig.4.6b) and their top constituent is FARG. 490 (7.8%) of the treebank sentence are FRAG. We create a rule-based logic to handle sentences with FRAG constituent. (1) We identify what part is missing. (2) The FRAG is replaced by a category that fits its missing part E.g for sentences that are missing subject $S \backslash NP$.



(a) Missing subject

(b) Missing object

(c) Missing Verb Phrase

Figure 4.6: Example of Partial Sentences with CCG Derivation

4.2.7 Questsions

The Hebrew language has questions words similar to English Wh-questions. A question sentence will usually end with the question sign ”?”. It is relatively easy to identify question sentence in written text. Question sentences are is always in SVO order, and in yes/no cases start with a question word. The Hebrew treebank has very few questions in it. Question sentence is marked with SQ instead of S.

4.3 Treebank Analysis

We use the standard train, dev and test partitioning of the Hebrew treebank.

Train 484-5,724

Dev 0-483

Test 5,724-6,216

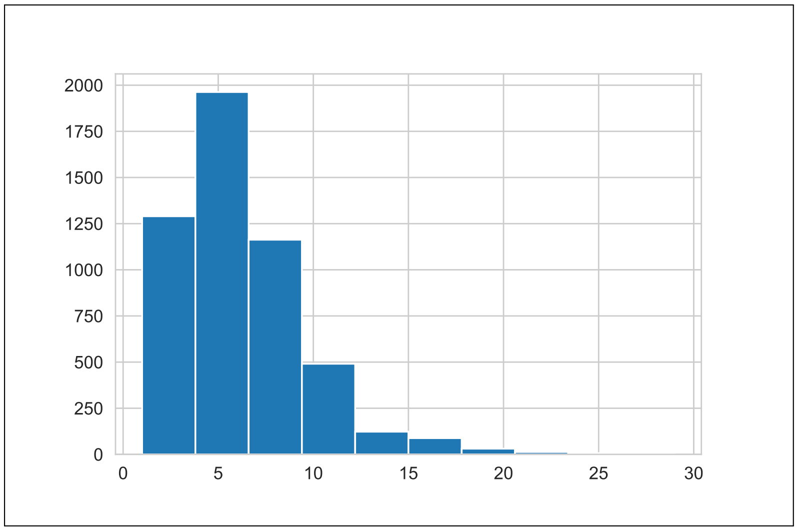


Figure 4.7: Sentences Length in the Hebrew treebank

The average sentence length is 17 words Fig. 4.7. The treebank has 1,500 categories: most appear only once and are ignored, 591 categories appear more than 3 times, accounting for 99.1% of all categories (figure 4.9), and are used in tagging and parsing. The most common category is NP. The average sentence contains 5 NP categories and about 23% of the morpheme (figure 4.9).

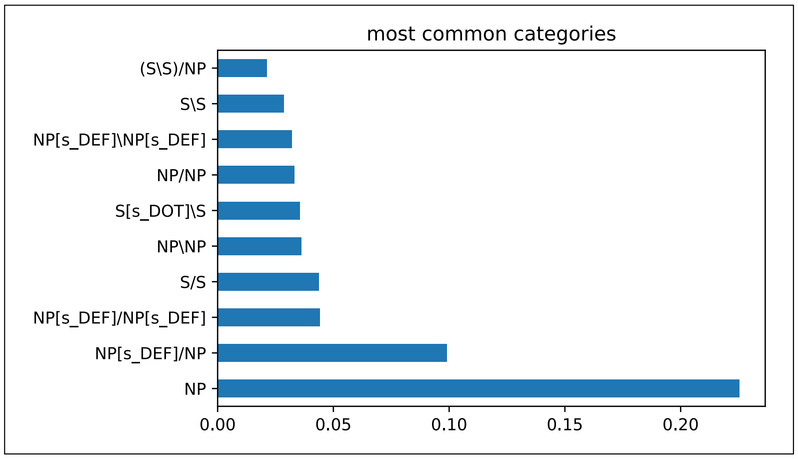


Figure 4.8: Most Common Categories

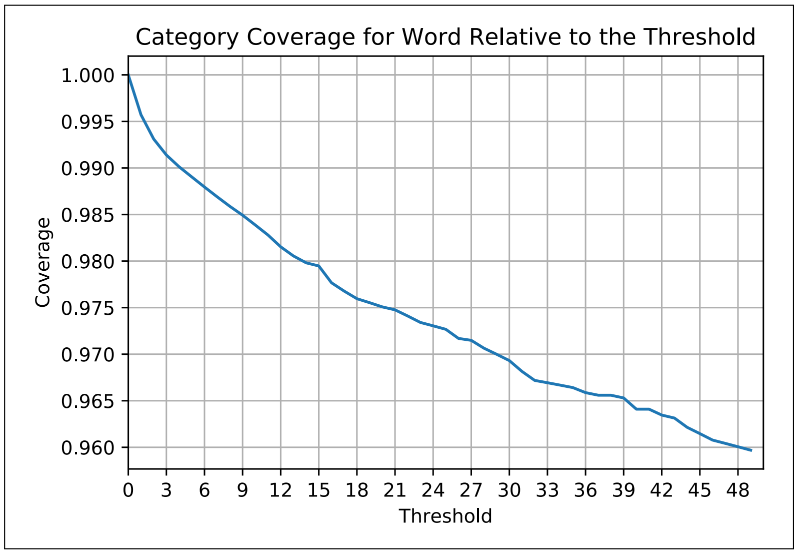


Figure 4.9: Minimum Category Occurrence Vs Coverage

Chapter 5

Semantic Parsing for Modern Hebrew

In the previous chapter, we discussed this thesis' first contribution - the treebank. In this chapter we will elaborate on the second contribution - a suite of parsing algorithms that we propose for CCG parsing of Hebrew. We will first lay out the experimental setup (5.1) and then describe the models that we propose. Section (5.2) focuses on the head-driven generative model for parsing CCG which we use as our baseline. In section (5.3) we discuss our models for parsing segmented morphological disambiguate texts. These models can be used with existing morphological disambiguation software. Finally, in section (5.4) we target morphological and semantic disambiguation, using models that jointly do the parsing and morphological disambiguation.

5.1 Experimental Settings

In this section we will give a short overview of the methods and data used in evaluation of our different models. We have 3 evaluation scenario's, supertagging gold segmented, supertagging unsegmented and finally parsing performance.

5.1.1 Datasplits

We used the treebank's traditional division into 3 sections: dev (0-483), train (484-5,723) and test (5,724-6,216). All the results from this chapter are from the test part of the Modern

Hebrew treebank. The development part was for parameters tuning, and the train part for model training.

5.1.2 Parsers

The A* Parser and Baseline Parser were implemented in java programing language. The A* is given the morphological lattice graph with each lattice, annotated with probability distribution for the tagger. When using gold segmentation the lattices is a linear graph.

5.1.3 Taggers

The CRF tagger was trained using CRFsuite [26]. We trained the Nerual network using Pytorch [27] framework, with SGD, momentum of 0.7, adaptive learning rate 0.1-0.001 and batch size of 32/64. We allowed up to 1000 epochs but used early stopping on the dev set. We used l2 regularization of 0.000001. We set dropout between 0.5 to 0.3 according to best dev set results. An unseen flag replaced categories that were filtered due to lower frequency in the training set. In prediction unseen flag is removed from the probability distribution.

5.1.4 Morphological disambiguation and Latice

The gold segmentation models use the morphemes from the Hebrew treebank. For BestMD model, We extracted the top 1 (best) morphemes from the raw sentences by using More A. [24] morphological disambiguation. We aligned the raw sentences with their matching trees using python difflib. Joint Model were fed lattice graph which We got by running More A. [24] Morphological analyzer.

5.1.5 Metrics

Tagger metrics were measured using the top 1 and top 20 accuracy of predicted categories. We counted categories missing from the model as errors.

ParseEval I did the ParseEval evaluation using the ParseEval tool by converting the CCG derivation tree as constituent trees.

Terminal The terminal evaluation metric counts the number of matching categories between parsed tree and gold tree. We note that the parse tree may contain different morphemes from the gold tree, in such cases we must first align the morphemes between the two trees. We used python difflib for morphemes alignment. We counted miss-aligned morphemes as errors. We measured the terminal accuracy in two settings: subcategorization (*NP[def]* != *NP]*) and without subcategorization (*NP[def]* = *NP]*)

Dependencies We extracted dependencies from aligned gold and parsed CCG trees. We aligned the morphemes of both sentences using difflib. Then assign minimum and maximum index for each node in the CCG derivation tree. These indexes were used to identify nodes between trees. The undirected dependencies where extracted by tracing the categories application and composition in both trees, resulting in a list of dependencies for each tree. We calculated Jaccard index to measure the number of capture dependencies.

5.2 Baseline

For our baseline performance, we implemented the Hockenmaier and Collins head base statistical model [13, 15]. let $t = e_1....e_n$ be a tree, where e_i is the i expansion, node, in the tree.

The Probability of the tree is given by the product of all the probability of each expansion.

$$P(t) = \prod_{e_i \in t} P(e_i) \tag{5.1}$$

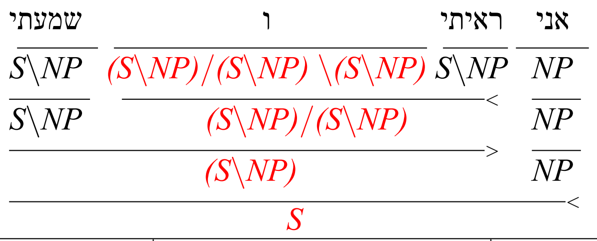
We define 3 expansion types: (1) Binary when a given node has two child node (2) Unary - when a given node has one child (3) Lexical when the given node is has a single leaf child. The binary expansion is calculated by estimating the conditional probability of the head node given the parent and expansion direction (left or right).

$$P_{e=binary}(p,h,o,d) \approx P(h|p) \times P(o|p,h,d) \tag{5.2}$$

$$P_{e=unary}(p,h,d) \approx P(h|p) \tag{5.3}$$

$$P_{e=lexial}(p,w) \approx P(w) \times P(w|c) \times P(c|p) \tag{5.4}$$

where p is the parent node, h is the head node, o is the other node (binary case), d is the direction of expansion, w is the morpheme text. Our baseline parser uses the CYK algorithm. The parse table is at the morpheme level and includes multiple derivations of each word. We decompose the words into lattices. We then use a lexicon to fill the possible categories for each morpheme in the lattice graph.



Step	**Rule**	**Head**	**Non-Head**	**Direcition**
ו ראיתי	head,non-head	((S\NP)/(S\NP))\(S\NP)	(S\NP)	backward
ו שמעתי	head,non-head	(S\NP)/(S\NP)	(S\NP)	forward
ו אני	head,non-head	(S\NP)	NP	backward
אני	lexicon, emission	NP	-	-
ראיתי	emission	(S\NP)	-	-
ו	emission	((S\NP)/(S\NP))\(S\NP)	-	-
שמעתי	emission	(S\NP)	-	-

Figure 5.1: Example of using the baseline probability model's growth rules on the sentence: " אניי ראיתי ו שמעתי " . Each line in the table corresponds to a single step in the CCG derivation of the sentence. We can see for each step which is the head, non-head and direction of application. These arguments are used by the 3 growth probabiliteis of the model we presented before: head, non-head and lexical (equations 5.2-5.4).
In the Figure we mark the head of each step in red.

The Hebrew treebank is relatively small, so effective smoothing is crucial. We use smoothing for both lexicon and tree expansion. Unseen morpheme receives the lexicon

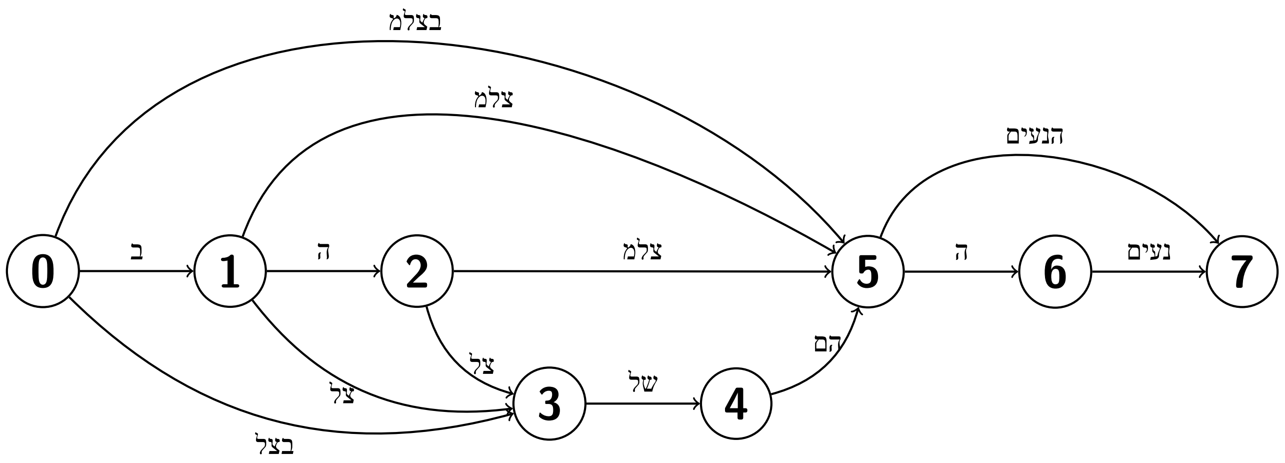


Figure 5.2: Example of 2 words sentences lattices decomposition, shown before in chapter 2.

entries that appear only once in our treebank. We also add these categories to entries of rare morphemes. For the expansion, we are using a rule that appears only one for unseen expansion both head and non-head. The conversion process is not without errors so we do not include categories that appear less than 3 times in the training set.

5.3 Segmented Text Supertagging

The Gold Segmentation model (figure 5.3) uses the treebank segmented sentences as input for its training. This model uses A* parser over the categories probability distribution. The supertagger annotates each morpheme with a probability distribution for the various categories. The A* parser [22] takes the morphemes and their probabilities. A* parser efficiently select the correct derivation. The A* parser sets upper and lower bound for each utterance. The boundaries are the sum of the best and worst probability for all the morphemes in the utterance.

5.3.1 Taggers

CRF Tagger

We trained a CRF tagger model using CRF suite [26]. The feature where the pos, gen, num and morpheme with 2 look-back and one look forward. The CRF Model the conditional probability of it's window feature on the category.

RNN Tagger

We trained a deep learning RNN similar to Zettlemoyer et. al.[21]. Our network had 3 layers Bi-direction GRU with 512 dim followings 3 fully connected relu layers. We use an embedding layer for the input morphemes. The network was trained using Statistical Gradient Descent in PyTorch. We used 0.1 learning rate and 0.7 momentum with a batch size of 64. We achieved 78% accuracy on our dev set. Our fully connected layers where 512 and 256.

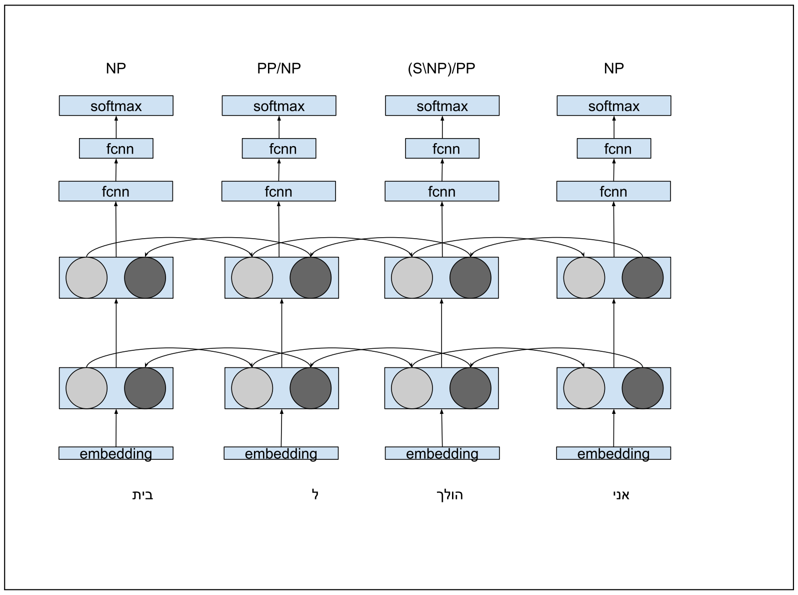


Figure 5.3: In the Figure, we see an example of the Morpheme BI-LSTM RNN Tagger activation. We feed the 4 morpheme long sentences into the model. We match each morpheme to its corresponding embedding. 2 Layers of bi-directional LSTM process, each embedding in the sentence context. We feed the resulting vectors into 2 fully connected layers with relu activation function. For the last layer we use Softmax activation to get probability distribution of the categories for a morpheme.

In practice, this model requires a different algorithm to select the best or K best segmentation for a sentence. The model greatest advantage is its simplicity since we can use an existing model for the English language. Selecting the best segmentation for a Hebrew utterance is effectively parsing. Any mistake in the K best would result in an error sentence. The search space is large. We will have to use a large K value to increase the chance that the correct decomposition is in them, which could easily result in the intractable model.

5.3.2 Results

The bi-directional LSTM had the best results with 77% accuracy on our test set. HMM had the worst results surprisingly even worse than the simple word model. More importantly, we can see that the top 20 categories contain the correct categories in more than 95% of the times.

Model	Accuracy	Top 20
HMM	37%	-
Word	57%	NA
CRF	71%	93%
RNN	77%	98%

Table 5.1: SuperTagger Accuracy when using Gold Segmentation

5.4 Joint Supertagging with Disambiguation

5.4.1 Taggers

Word-Context

The word-context (Fig. 5.4) inputs model take the words text plus a list of possible morphemes decomposition of the word. The model then outputs a category probability distribution for each morpheme. The model performs both the tagging and the morpheme disambiguation in a single run. The network architecture is composed of 2 Layers bi-directional GRU followed by 3 layers of fully connected relu. The word and morpheme share the same encoding layers as we found it works better. The words embedding is passed to the GRU layers. The GRU hidden state of each word is then concatenated with the embedding of each morpheme in the word decomposition. The concatenated word morpheme vector is then processed in the fully connected layers. The network outputs a softmax probability of categories for each morpheme. The network is trained on the gold treebank with a known segmentation. We used a similar SGD with 0.1 learning rate and 0.7 momentum to train the network. In evaluation time we use YAP [24] morphological analyzer to decompose the words into morphemes. We use pytorch framework [27] to train our NN models .

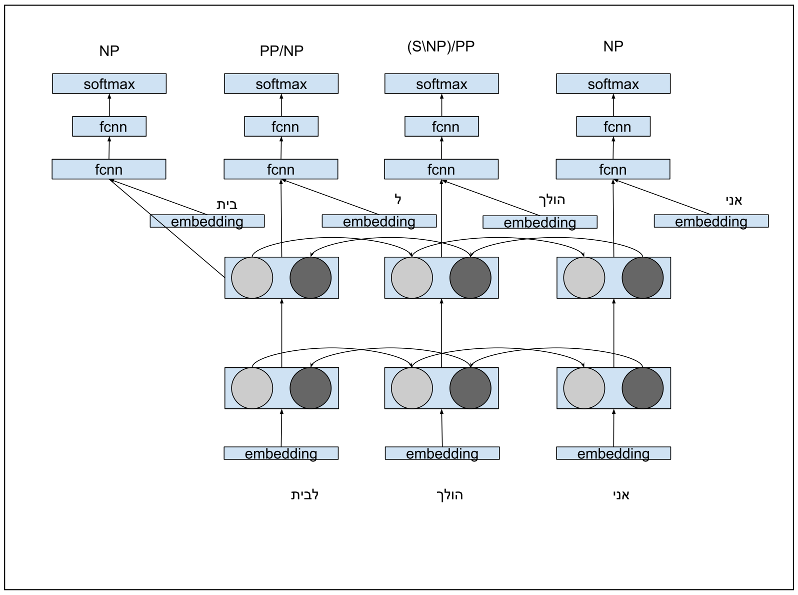


Figure 5.4: I
llustration Word-Context Super Tagger] The word-context model assumes independence between the selected morpheme categories both in the word and between words. The assumption is obviously not true and hurt the model performance, We tried to amend it by conditioning on the previous select composition.

Manual-Morphemes Encoding

We also want to try if we can build a better embedding for both words and morpheme by including separate morphological features. We expected our encoding to work better since the Hebrew treebank is relatively small and many words are too sparse for a model to estimate their hidden morphology. We created a separate representation for each morphological feature. We choose the dimension of each morphological feature by the number of possible values it has. We used a separate embedding layer for each feature and we added a fully connected layer before the LSTM. The region encoder, uses a constant 9 dim vector to encode a word. The regions are: PRP, AT, 1, 2, H, IN, POS, REL and CC. It maps each morpheme in the word to one of these regions. Unknown morphemes are mapped to 1 if two unknown morphemes are found, then the first would be mapped to 1 and the second to 2.

Lattice-Morphemes-Encoding

The Lattice-Morphemes-Encoding (figure 5.5) labels each word by concatenating all of its morpheme categories. The network architecture is similar to the post segmentation model. We are using wider layers since we transfer significantly more information. We also use 3 GRU layers instead of 2 layers as we found it work better.

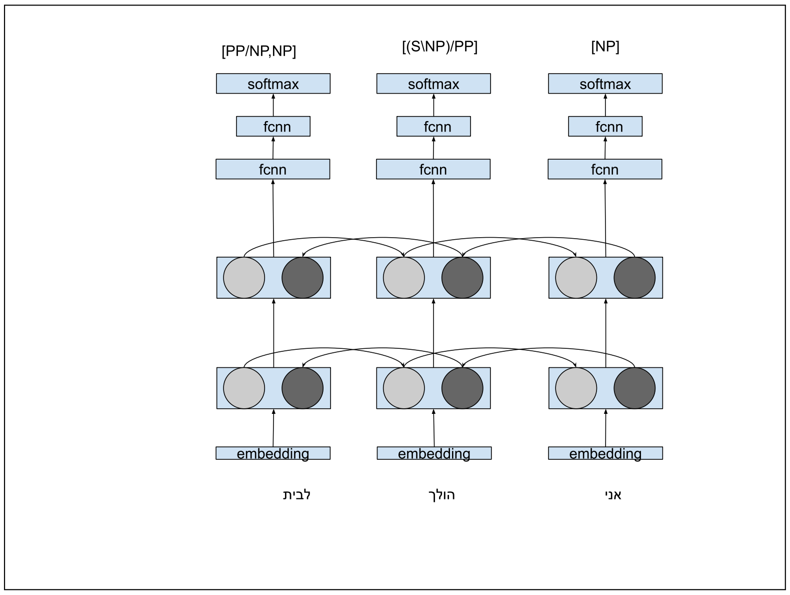


Figure 5.5: In the figure, we see an example of the lattice-encoding BI-LSTM RNN Tagger activation. We feed the 4 morpheme/3 words long sentences into the model. Each word is represented by a path in the lattice graph, the output for each word is a list of categories tuples. In Each tuple we have one category per morpheme.

5.4.2 Results

Model	Accuracy	Top 15
Best MD + Tagger	56%	NA%
Morpheme-Manual-Encoding	58%	95.5%
Morpheme-Super-State	42%	83%
Words-Context-Morphemes	64%	97%

Table 5.2: SuperTagger Accuracy Joint Segmentation

5.5 Parsing Results with A*

The tagged morphemes were passed into an A* parser. The A* parser sets an upper and lower boundary for each constituent by multiplying the best and worst probabilities by its child constituents. The performance of A* parser depends on the supertagger's accuracy. The English supertagging has reached over 91% accuracy. Unfortunately, our best tagger was only at 63% which make A* parser questionable.

We preformed an evaluation of our parser performance by measuring it. We measured performance with three metrics: ParseEval, Terminal Accuracy and Undirected Dependencies. ParseEval roughly measures the amount of constituents we assigned a correct category by the parser. We note that CCG tree might have a different derivation order which are equivalent from CCG point of view but might get a different ParseEval score. The Terminal Accuracy measures how many of the morphemes were assigned the correct category. A more effective metric is the Dependency Score. Our Terminal Accuracy is significantly better than our dependency's accuracy. Indicating our parser needs better modeling for the selection of attachments. The evaluation of the parsers shows there is room for progress. The word-context model was the best tagging model by a considerable margin. We were surprised by the result since we expected the lattice encoding to preform better due to its use in pos tagging. There are around 30 pos tags and 3 few words with over 3 morphemes. While there are over 500 categories so our lattice encoding model has significantly more target space which explain the gap in performances.

Model	ParseEval	Terminal	Unidirect dependecies
GoldSegmentation	28%	44% | 70%*	44%
BestMD + Word	21%	29% | 51%*	39%
Baseline	13%	19% | 25%*	20%
Words-Context-Morphemes	22%	42% | 58%*	35%

Table 5.3: The parsing performances of our parsers. We extract dependencies from the CCG tree by identifying the arguments of functional categories. We calculate the accuracy based on the number of dependencies in the gold and parse divided by the union of both. Terminal accuracy measure how many morphemes received the correct categories from the parser.

5.6 Analysis and Conclusion

In this chapter we have presented our results for supertagging for both segmented and unsegmented text and how they reflect on the parsing performance. Table 5.2 contains the results of our supertagging models on the treebank test. We used the treebank's morphological segmentation so the model performs only tagging. We can see that the RNN based model achieved the best accuracy, similar to finding in other languages. Supertagging on segmented text is not realistic, but can be used to give us a good estimation of loss of accuracy due to morphological disambiguation.

In Table 5.2 we can see the result for unsegmented text. Again, the results are on the treebank test set, but this time on the raw sentences without segmentation. The Word-Context model preformed better in terminal accuracy compared to the BestMD. This result is consistent with previous works that show joint parsing performs better than separate disambiguation phase. we we can see 13% drop in accuracy between the best unsegmented to the best segmented models. There is clearly room for better a joint disambiguation strategy in supertagging.

The top 1 result accuracy is significantly lower than the top 15 for both the segmented and unsegmented cases. This tells us that our supertagger is good in selecting the most probable 15 categories (among the 300) but has failed to select the best one from this short list of 15 categories.

Table 5.3 contains the parsing results for our different parsing models. We can see is a correlation between tagging and parsing performance. The terminal accuracy reflects the accuracy of the parser in capturing the correct category while the undirected dependencies capture the accuracy of the parser in relation between words. The Parseval is less relevant since multiple CCG derivation tree, which represent the same relation but in a different order, will be counted as error. One of the main reason for the drop in performance is incorrect selection of verb categories for verbs. These errors result from missing categories or incorrect estimation of an existing category for unseen verbs. These errors will propagate to the verb arguments and modifiers.

Chapter 6

Conclusion

In this thesis our goal was to create the first treebank and parser for semantic parsing for Hebrew. We chose to base our formal representation and algorithm on the formal notions of CCG. The contribution of the paper are threefold: (1) we created a new Hebrew CCG benchmark consisting of 6,221 sentences annotated with CCG trees, (2) we proposed different parsing architectures to deal with the close interaction between CCG structures and morphology in the Hebrew data, and (3) we presented baseline results for Hebrew semantic parsing. Moreover we developed a web based tool for viewing our CCG treebank which can in future development of the treebank and parsers. All of our data, models and code are made publicly available to the community at the OnlpLab reportory.

We created a CCG treebank for the Hebrew language. To achieve this, we developed a new conversion process for the Hebrew language based on Hocknmier's [13] process. We evaluated 3 different approaches for parsing: (1) head-driven model, (2) best MD, and (3) word context; and 3 different approached for joint supertagging: (1) lattice encoding, (2) regional encoding, and (3) word context. The intent was to apply supertagging on lattice graphs. In the process of developing the treebank we created a web interface for viewing and analyzing our CCG treebank.

The treebank construction is still an ongoing effort due to the lack of coverage of some dependencies type. The number of categories relative to the treebank's size is high and many of them appear less frequently. Many of these categories are results of a rare word order sequence. It is worth checking the possiblity of utilizing larger datasets, or to normalize verb category to a single word-order.

The number of morphemes with NP category seems too high, so it might be beneficial splitting it into multiple categories, however that could cause creation of additional verb categories. A possible solution is to model morphological features such as agreement and gender outside the CCG categories, or as optional subcategories that do not appear on the verb.

The parsing performance shows a strong correlation to the tagger performance, so increasing the tagging performance should should lead to better parsing results on the whole. We suggest considering adding more features to in order to increase tagging performance.

The word-context model assumes independence between the selected morpheme categories in both the word itself and in between words. This model's assumption means that it might miss agreement information between words, yet it still produced the best results. We need to find a good strategy to run a supertagger on lattice graph.

Finally, we hope that others can use this treebank and parsers to further improve NLP for Modern Hebrew, a Semitic, morphologically-rich and resource-scarce language.

Bibliography

[1] Anthony E. ”Ades and Mark J.” Steedman. “On the order of words”. In: *Linguistics and Philosophy* 4.4 (Dec. 1982), pp. 517–558. issn: 1573-0549. doi: 10.1007/BF00360804. url: https://doi.org/10.1007/BF00360804.

[2] Omri Abend and Ari Rappoport. “The State of the Art in Semantic Representation”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 77–89. doi: 10.18653/v1/P17-1008. url: https://www.aclweb.org/anthology/P17-1008.

[3] Omri Abend and Ari Rappoport. “UCCA: A Semantics-based Grammatical Annotation Scheme”. In: *Proceedings of the 10th International Conference on Computational Semantics (IWCS 2013) – Long Papers*. Potsdam, Germany: Association for Computational Linguistics, Mar. 2013, pp. 1–12. url: https://www.aclweb.org/anthology/W13-0101.

[4] Omri Abend and Ari Rappoport. “Universal Conceptual Cognitive Annotation (UCCA)”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, pp. 228–238. url: https://www.aclweb.org/anthology/P13-1023.

[5] Meni Adler and Michael Elhadad. “An Unsupervised Morpheme-Based HMM for Hebrew Morphological Disambiguation”. In: *ACL*. Ed. by Nicoletta Calzolari, Claire Cardie, and Pierre Isabelle. The Association for Computer Linguistics, 2006. url: http://dblp.uni-trier.de/db/conf/acl/acl2006.html#AdlerE06.

[6] Laura Banarescu et al. “Abstract Meaning Representation for Sembanking”. In: Aug. 2013, pp. 178–186.

[7] Srinivas Bangalore and Aravind K. Joshi. “Supertagging: An Approach to Almost Parsing”. In: *Comput. Linguist.* 25.2 (June 1999), pp. 237–265. issn: 0891-2017. url: http://dl.acm.org/citation.cfm?id=973306.973310.

[8] Roy Bar-Haim, Khalil Sima’an, and Yoad Winter. “Part-of-speech tagging of Modern Hebrew text”. In: *Natural Language Engineering* 14 (Apr. 2008), pp. 223–251. doi: 10.1017/S135132490700455X.

[9] Yehoshua Bar-Hillel. “A Quasi-Arithmetical Notation for Syntactic Description”. In: *Language* 29.1 (1953), pp. 47–58. issn: 00978507, 15350665. url: http://www.jstor.org/stable/410452.

[10] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. issn: 2307-387X.

[11] Stephen A. Boxwell and Chris Brew. “A Pilot Arabic CCGbank”. In: *LREC*. 2010.

[12] Ruken Cakici. “Automatic Induction of a CCG Grammar for Turkish”. In: *ACL*. 2005.

[13] Combinatory Categorial Grammar and Julia Hockenmaier. “Data and Models for Statistical Parsing with Combinatory Categorial Grammar”. Doctoral dissertation. Sept. 2003.

[14] Stephen Clark and James R. Curran. “Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models”. In: *Computational Linguistics* 33.4 (2007), pp. 493–552. doi: 10.1162/coli.2007.33.4.493. url: https://doi.org/10.1162/coli.2007.33.4.493.

[15] Michael Collins. “Head-Driven Statistical Models for Natural Language Parsing”. In: *Computational Linguistics* 29.4 (2003), pp. 589–637. doi: 10.1162/089120103322753356. eprint: https://doi.org/10.1162/089120103322753356. url: https://doi.org/10.1162/089120103322753356.

[16] Yoav Goldberg. “Automatic Syntactic Processing of Modern Hebrew”. Doctoral dissertation. 2011.

[17] Yoav Goldberg and Reut Tsarfaty. “A Single Generative Model for Joint Morphological Segmentation and Syntactic Parsing”. In: *Proceedings of ACL-08: HLT*. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 371–379. url: https://www.aclweb.org/anthology/P08-1043.

[18] Julia Hockenmaier. “Creating a CCGbank and a Wide-coverage CCG Lexicon for German”. In: *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*. ACL-44. Sydney, Australia: Association for Computational Linguistics, 2006, pp. 505–512. doi: 10.3115/1220175.1220239. url: https://doi.org/10.3115/1220175.1220239.

[19] Alon Itai and Yoad Winter. “Building a Tree-Bank of Modern Hebrew Text”. In: (Oct. 2001).

[20] Aravind K. Joshi and B. Srinivas. “Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing”. In: *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*. COLING ’94. Kyoto, Japan: Association for Computational Linguistics, 1994, pp. 154–160. doi: 10.3115/991886.991912. url: https://doi.org/10.3115/991886.991912.

[21] Mike Lewis, Kenton Lee, and Luke Zettlemoyer. “LSTM CCG Parsing”. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016, pp. 221–231. doi: 10.18653/v1/N16-1026. url: https://www.aclweb.org/anthology/N16-1026.

[22] Mike Lewis and Mark Steedman. “A* CCG Parsing with a Supertag-factored Model”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 990–1000. doi: 10.3115/v1/D14-1107. url: https://www.aclweb.org/anthology/D14-1107.

[23] Tomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems* 26 (Oct. 2013).

[24] Amir More and Reut Tsarfaty. “Data-Driven Morphological Analysis and Disambiguation for Morphologically Rich Languages and Universal Dependencies”. In: *Proceedings of COLING 2016*. Osaka, Dec. 2016.

[25] Hiroki Nakayama et al. *doccano: Text Annotation Tool for Human*. Software available from https://github.com/doccano/doccano. 2018. url: https://github.com/doccano/doccano.

[26] Naoaki Okazaki. *CRFsuite: a fast implementation of Conditional Random Fields (CRFs)*. 2007. url: http://www.chokkan.org/software/crfsuite/.

[27] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. url: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[28] Jeffrey Pennington, Richard Socher, and Christopher Manning. “Glove: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162. url: https://www.aclweb.org/anthology/D14-1162.

[29] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Chicago: The University of Chicago Press, 1994.

[30] Carl Pollard and Ivan A. Sag. *Information-based Syntax and Semantics: Vol. 1: Fundamentals*. Stanford, CA, USA: Center for the Study of Language and Information, 1988. isbn: 0-937073-24-5.

[31] Rion Snow et al. “Cheap and Fast – But is it Good? Evaluating Non-Expert Annotations for Natural Language Tasks”. In: *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. Honolulu, Hawaii: Association for Computational Linguistics, Oct. 2008, pp. 254–263. url: https://www.aclweb.org/anthology/D08-1027.

[32] M. Steedman. *Surface Structure and Interpretation*. Linguistic inquiry monographs. MIT Press, 1996. isbn: 9780262691932. url: https://books.google.co.il/books?id=Mh1vQgAACAAJ.

[33] Martin Sundermeyer, Hermann Ney, and Ralf Schluter. “From Feedforward to Recurrent LSTM Neural Networks for Language Modeling”. In: *Audio, Speech, and Language Processing, IEEE/ACM Transactions on* 23 (Mar. 2015), pp. 517–529. doi: 10.1109/TASLP.2015.2400218.

[34] Ahmed I. El-taher et al. “An Arabic CCG approach for determining constituent types from Arabic Treebank”. In: 2014.

[35] Stephen Tratz. “Dependency Tree Annotation with Mechanical Turk”. In: Jan. 2019, pp. 1–5. doi: 10.18653/v1/D19-5901.

[36] Reut Tsarfaty. “A Unified Morpho-Syntactic Scheme of Stanford Dependencies”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, pp. 578–584. url: https://www.aclweb.org/anthology/P13-2103.

[37] Reut Tsarfaty. “Relational-Realizational Parsing”. Doctoral dissertation. 2010.

[38] Reut Tsarfaty et al. “What’s Wrong with Hebrew NLP? And How to Make it Right”. In: Jan. 2019, pp. 259–264. doi: 10.18653/v1/D19-3044.

[39] Wenduan Xu, Michael Auli, and Stephen Clark. “CCG Supertagging with a Recurrent Neural Network”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 250–255. doi: 10.3115/v1/P15-2041. url: https://www.aclweb.org/anthology/P15-2041.

תקציר

ניתוח מחרוזות סמנטי הינה טכנולוגיית מפתח אשר משמשת למיפוי פסוקיות של שפה טבעית לייצוג פורמלי של משמעותן. הטכנולוגיה מיושמת במספר רב של יישומי עיבוד שפה טבעית מודרניים כמו "אלקסה" של אמזון, "סירי" של אפל ועוד.

Combinatorial Categorical Grammar (CCG) הינה שיטה המשמשת לפיתוח ניתוח מחרוזות סמנטי. Treebanks ומנתחי מחרוזות בשיטת CCG נוצרו עבור שפות רבות, אך מעולם לא פותחו עבור שפות שמיות. מטרתה של עבודה זו הינה ליצור את ה Treebank ומנתחי המחרוזות הראשונים בשיטת CCG לשפה העברית המודרנית.

אנו משתמשים ב constituent treebank בעברית כנקודת מוצא ליצירת ה CCG Treebank בעברית. אנו מציעים תהליך המרה דומה לזה שהציע [13] ליצירת ה CCGBank באנגלית.

להמרה שלנו ישנם 3 שלבים: (1) קביעת סוגו של כל מקטע. (2) המרת עץ המקטעים לעץ בינארי. (3) קביעת קטגוריית CCG לכל מקטע, בהתבסס על מיקומו של המקטע בעץ הבינארי, תפקידו הסמנטי, התיוגים השייכים לו וסוג ה *part-of-speech (POS)* של המקטע. ה Treebank העברי מורכב מתיוגים של צורנים, לא מילים. צורן הינה היחידה הלשונית הקטנה ביותר הנושאת משמעות. בשפות העשירות בצורנים כמו עברית, מילה אחת מורכבת ממספר רב של צורנים וכל צורן מחזיק בקטגוריית CCG משלו. מנתח המחרוזות הבסיסי שאנו מציעים דומה למנתח שהוצע על ידי *Hockenmaier (head-driven)* .

אנו מציעים שימוש מבוסס suppertagging כיוון שהוכח שזה מניב תוצאות מוצלחות בשפות אחרות. ביצוע suppertagging על טקסט בעברית מצריך פירוק צורני של מילים. בדקנו 3 שיטות ליישום sup-pertagging בשילוב פירוק צורני. (1) הפעלת RNN ברמת המילים על מנת לקבל וקטור מייצג לכל מילה, בסיום הליך זה, המרכיבים הצורנים של כל מילה מועברים דרך שכבת embedding לקבלת וקטור מייצג לכל צורן. הוקטורים שנתקבלו מאוחדים יחד. (2) ייצור גרף צורני לכל מילה, כאשר כל נתיב מורכב מהצורנים שבתוכו. הנתיבים מועברים דרך שכבת embedding לקבלת וקטור מייצג לכל מילה. (3) יצירת קידוד ידני ייחודי לכל מילה על פי סוגי הצורנים המרכיבים אותה.

המודל הראשון (המאחד את הוקטורים של המילים עם אלה של הצורנים) תפקד באופן הטוב ביותר. מודל הקידוד הידני השיג את התוצאות הטובות ביותר אחריו. המודל הגרפי תפקד באופן פחות טוב עקב מרחב החיפוש העצום. אנו מסיקים כי ניתוח צורני מקדים ל suppertagging מועיל, לכל הפחות עבור ה CCG treebank במאגר הקטן של השפה העברית.

עבודה זו בוצעה בהדרכתה של פרופ׳ רעות צרפתי (אוניברסיטת בר-אילן) ובעזרתו של ד״ר אילן גרונאו (בי״ס אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה)

המרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.) - מסלול מחקרי

דקדוק קטגורי קומבינטורי לשפה העברית

מאת
ארז אגמי

עבודת תזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.
במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

2020 28, June